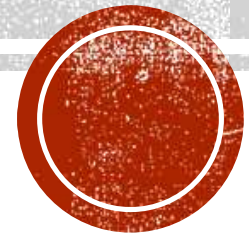


WEBSERVICES

History





TECH

New Jersey needs volunteers who know COBOL, a 60-year-old programming language

PUBLISHED MON, APR 6 2020-6:05 PM EDT | UPDATED MON, APR 6 2020-7:44 PM EDT

Kif Leswing
@KIFLESWING

SHARE

KEY POINTS

- If you know how to code COBOL, the state of New Jersey wants to hear from you.
- Systems that power unemployment benefits in New Jersey are running off of 40-year-old mainframes that require COBOL
- New Jersey plans to ask for volunteers with a variety of skills, including technologists



A man using a computer in the 1950s.
Getty Images, Ulfstein Bild

If you know how to code COBOL, the state of New Jersey wants to hear from you.

New Jersey Gov. Phil Murphy says that the state is looking for volunteers with skills that can be used to help in the COVID-19 [coronavirus](#) outbreak, and one of those skills is knowing your way around a 61-year-old programming language used on big, old, mainframe computers.

TV

Squawk Box [WATCH LIVE](#)

UP NEXT: [Squawk on the Street](#) 9:00 AM ET [Listen](#)

TRENDING NOW

- Omicron-specific vaccine is coming, but it 'may not matter—everybody's going to be infected': expert
- Goldman's David Kostin says a tech disconnect is the 'single greatest mispricing' in U.S. stocks
- Pfizer CEO says two Covid vaccine doses aren't 'enough for omicron'
- Here's the deadline for your 2021 tax return
- Government may scale back Medicare Part B premium increase

Sponsored Link by Taboola



WE NEED COBOL
PROGRAMMERS FOR
OUR MAINFRAME
MILLENIUM PROBLEM.

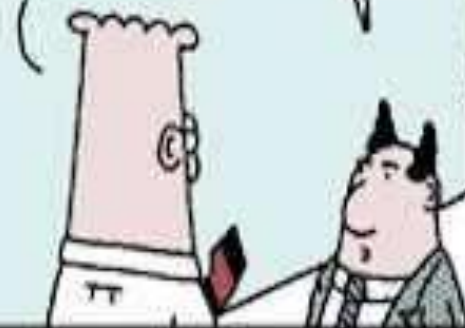


www.unitedmedia.com

S. Adams

IF YOU SEE ANYONE
WHO LOOKS LIKE
A COBOL PROGRAMMER,
LET ME KNOW.

TURN
AROUND.



11/4/97 © 1997 United Feature Syndicate, Inc.

ARE YOU A COBOL
PROGRAMMER?

NO, BUT I'M OFTEN
TOLD I LOOK LIKE
ONE.

YOU'RE
HIRED.



COMPONENT ARCHITECTURES

- A component architecture is a method of designing software components
 - The components should be easily connected together, reused, or replaced
 - Without re-compiling the application that uses them.
- A Component, is a piece of software that:
 - Is like a library, rather than a stand-alone application
 - Is distributed in a compiled, executable form
 - Exposes a group of methods and properties to its client application



HISTORICAL APPROACHES

1. CORBA (OMG)
2. DCOM (Microsoft)
3. XPCOM (Mozilla)
4. RMI (Sun Microsystems) on JRMP/IIOP



COMPUTER



SOFTWARE

COMPUTER



SOFTWARE-1

COMPUTER

SOFTWARE-2





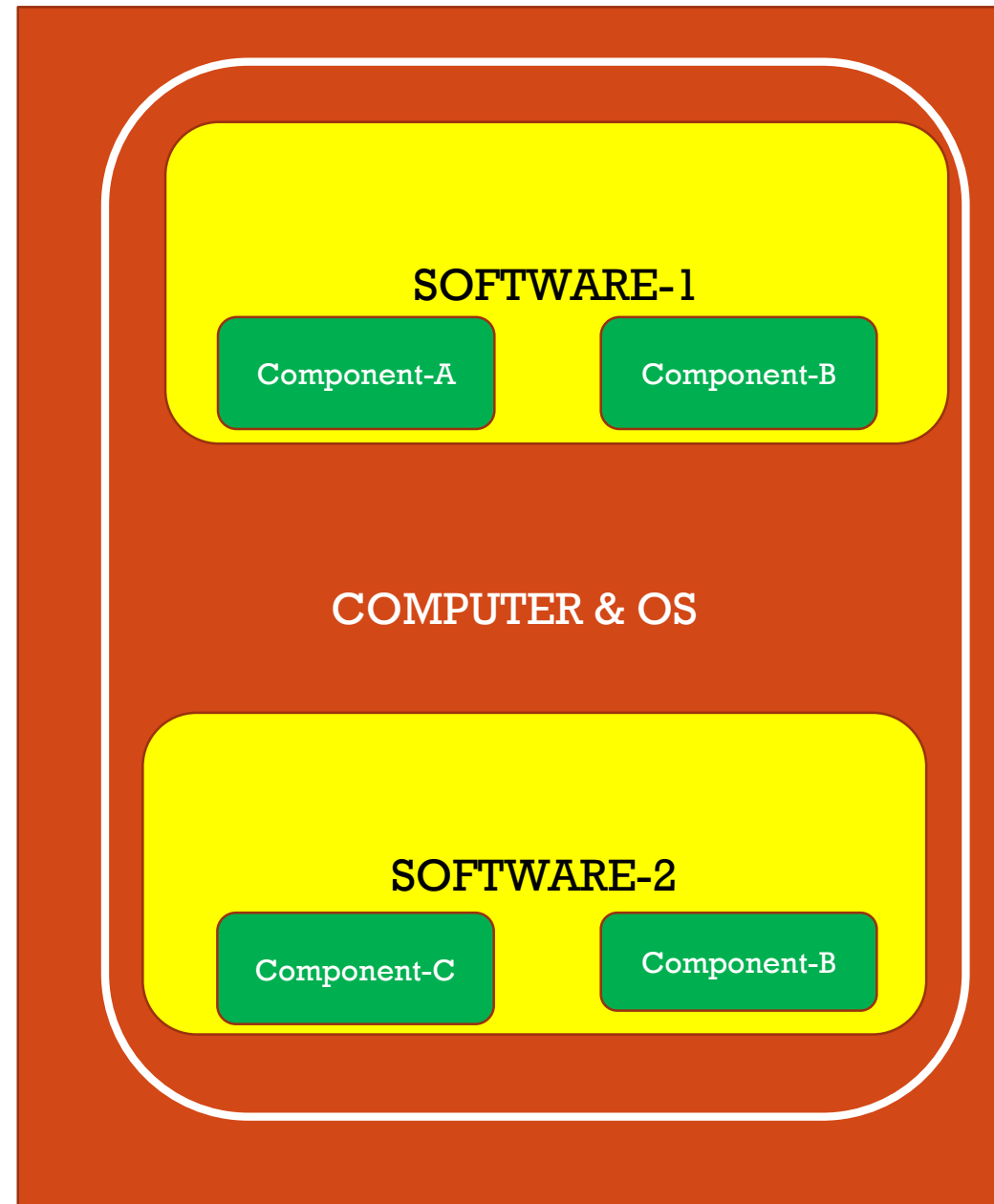
A diagram showing a stack of software layers. It consists of a large orange rectangle containing a white rounded rectangle. Inside the white rectangle are two yellow rounded rectangles, one at the top and one at the bottom. The text 'SOFTWARE-1' is centered in the top yellow rectangle, 'COMPUTER & OS' is centered in the white rectangle, and 'SOFTWARE-2' is centered in the bottom yellow rectangle.

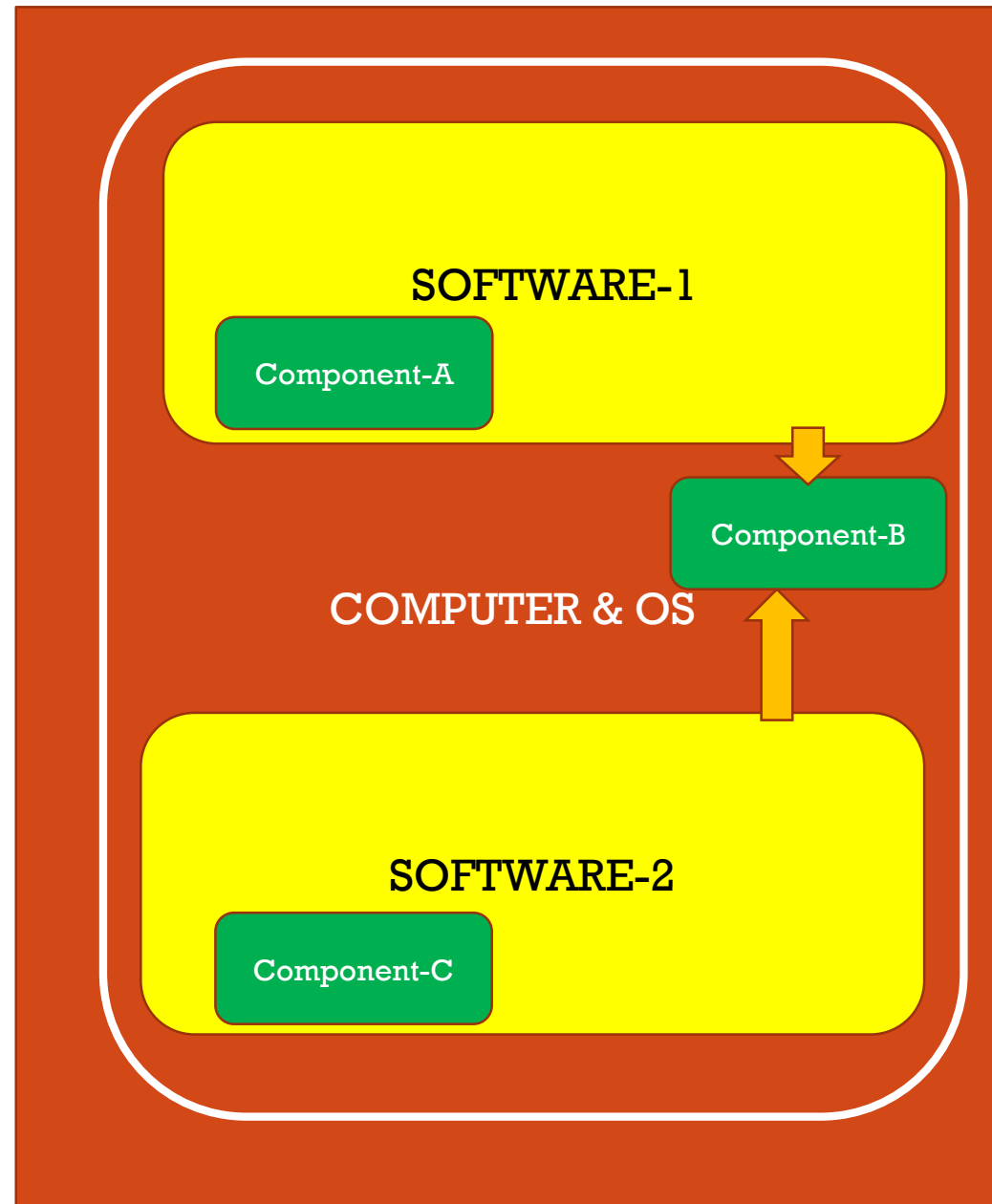
SOFTWARE-1

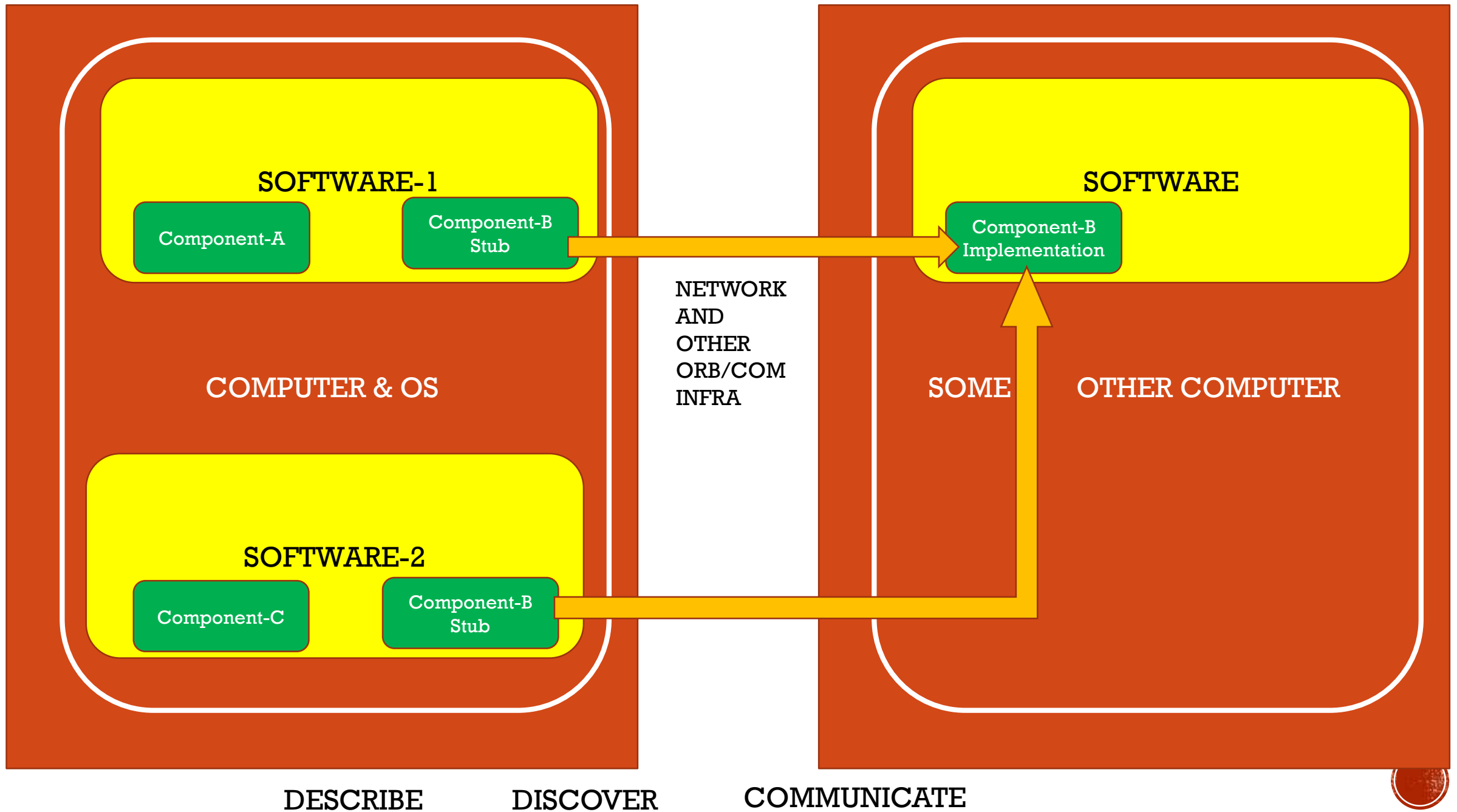
COMPUTER & OS

SOFTWARE-2





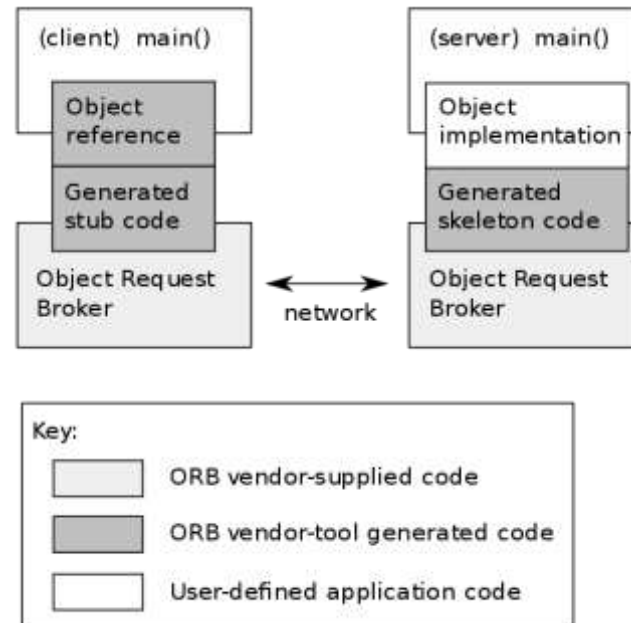




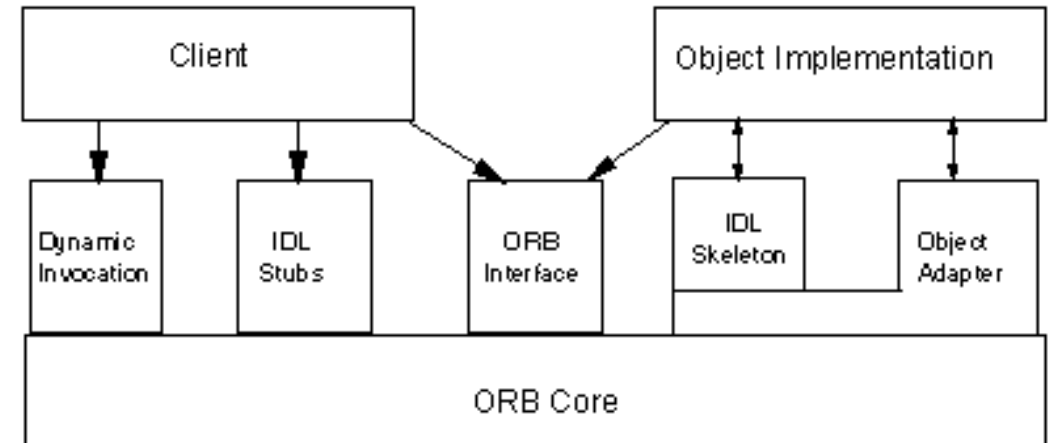
CORBA



- CORBA (OMG) It is standards-based, vendor-neutral, and language-agnostic. Very powerful but limited however by its complicated way of utilizing the power and flexibility of the Internet.



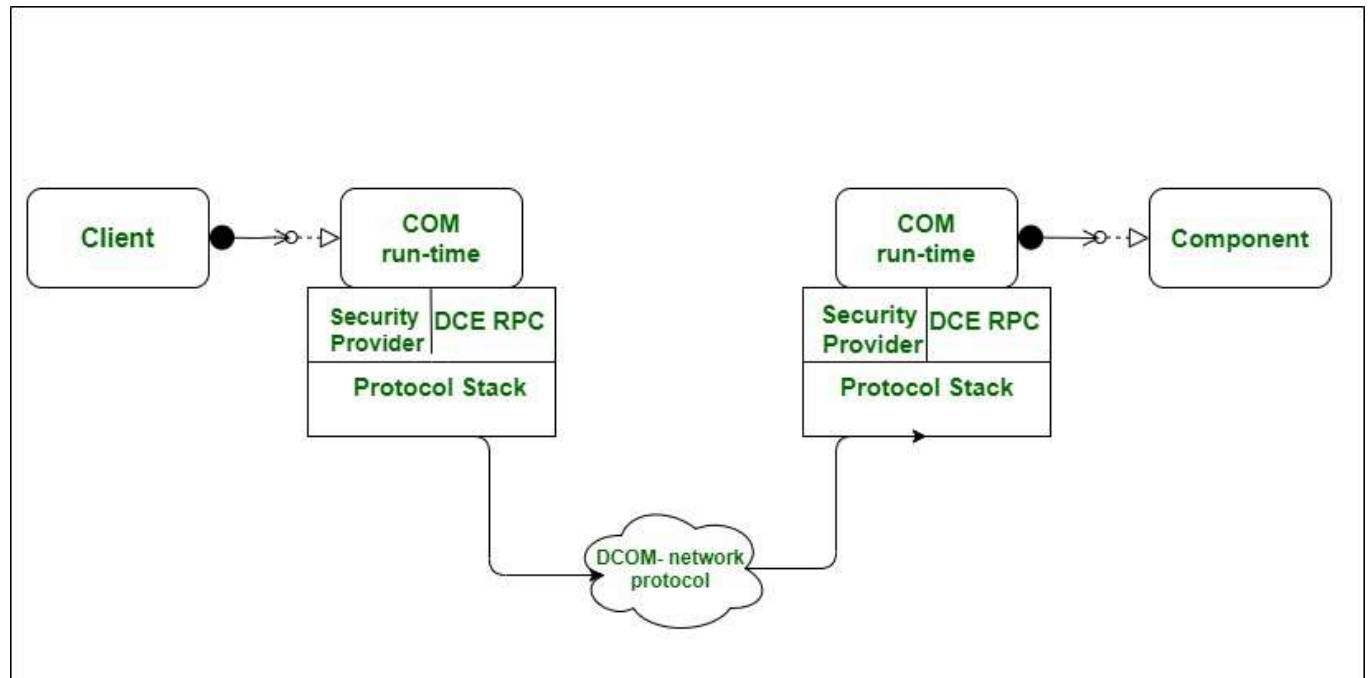
CORBA Architecture



DCOM



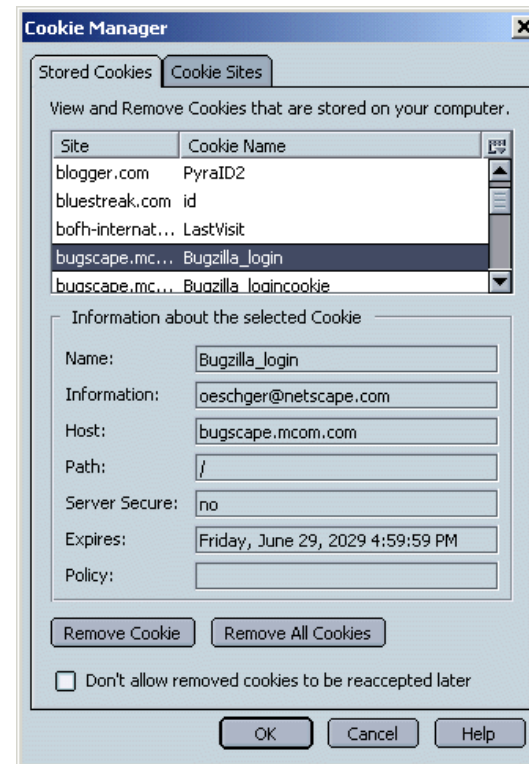
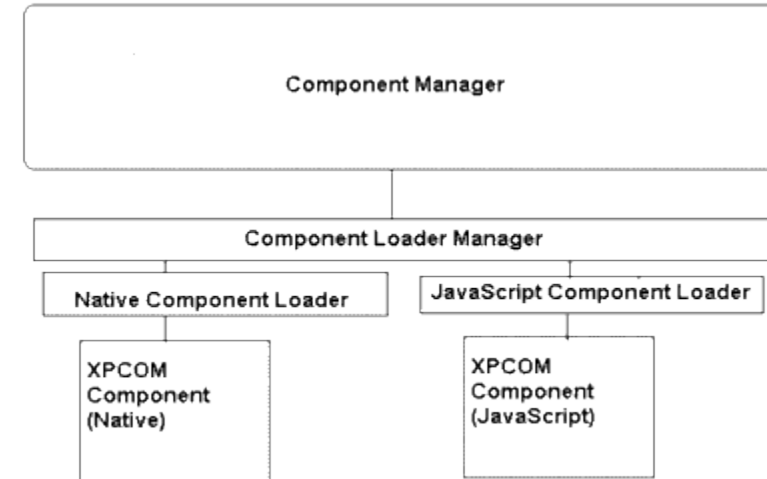
- DCOM (Microsoft) Distributed Computing platform closely tied to Microsoft component efforts such as OLE, COM and ActiveX.



XPCOM



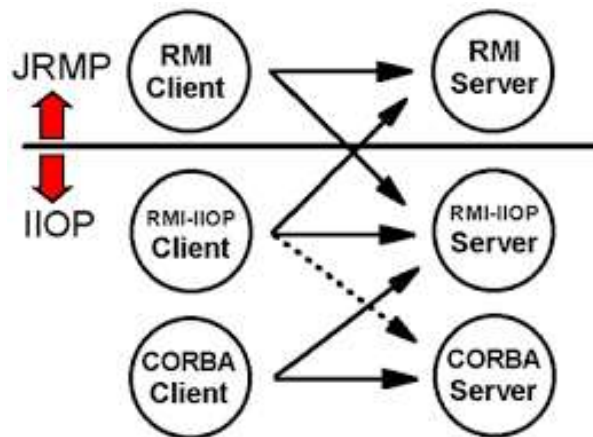
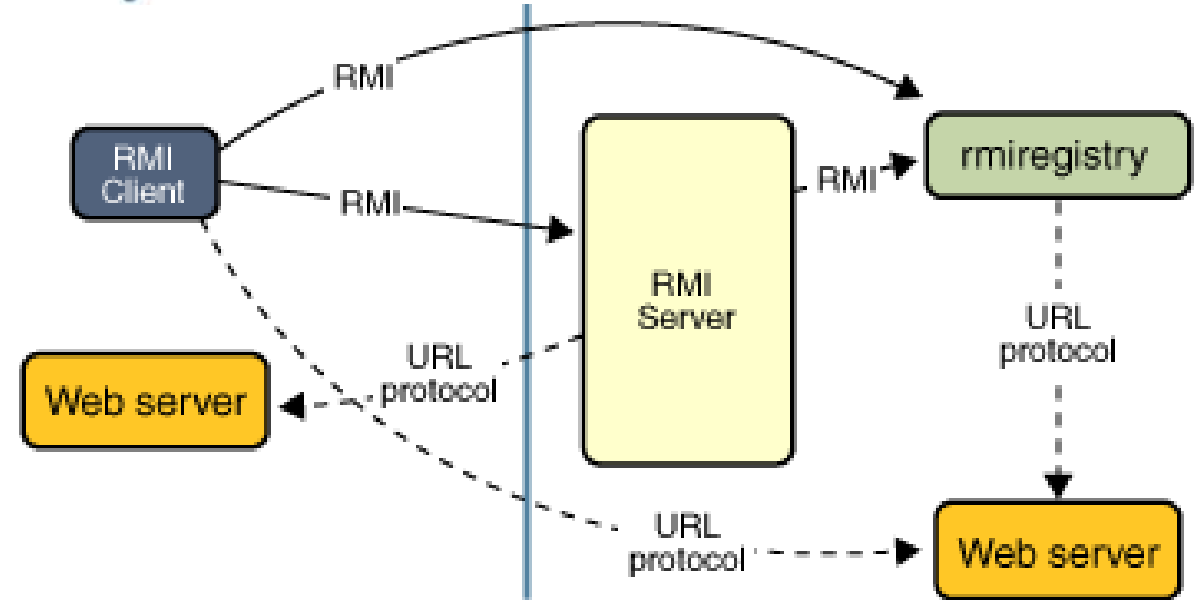
- XPCOM is a Mozilla made competitor to DCOM (Microsoft). Stands for Cross.



JAVA - RMI



- Java RMI provides the mechanism by which the server and the client communicate and pass information back and forth. This build a distributed object application.
- The RMI over IIOP implementation supports interop with CORBA



HISTORICAL APPROACHES (MORE TO READ)

- IDL: <https://www.omg.org/spec/IDL/About-IDL/>
 - Example: <http://jmvidal.cse.sc.edu/csce590/spring02/corba-idl-intro.pdf>
- DCOM (Microsoft) :
 - (COM) <https://www.cs.umd.edu/~pugh/com/>
 - <https://condor.depaul.edu/elliott/513/projects-archive/DS420Fall98/Edinburgh/dcom.htm>
 - XPCOM (Mozilla) : <https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM>
- CORBA (OMG) It is standards-based, vendor-neutral, and language-agnostic. Very powerful but limited however by its complicated way of utilizing the power and flexibility of the Internet. : <https://www.omg.org/spec/CORBA/About-CORBA/>
- RMI (Sun Microsystems) <https://docs.oracle.com/javase/tutorial/rmi/overview.html>
- Web Services (W3C) Web services **are more of an evolution than a revolution**



WHAT IS A WEBSERVICE?

Definition: A Web Service is a standards-based, language-agnostic software entity, that accepts specially formatted requests from other software entities on remote machines via vendor and transport neutral communication protocols, producing application specific responses.

- Standards based - W3C, etc
- Language agnostic - Any programming language
- Formatted requests - XML, JSON, EDN, BSON etc
- Remote machines – Over the network (web)
- Vendor neutral – No one vendor controls the standards or specifications
- Transport neutral – HTTP, SMTP, ??
- Application specific responses – *Business-aware* responses



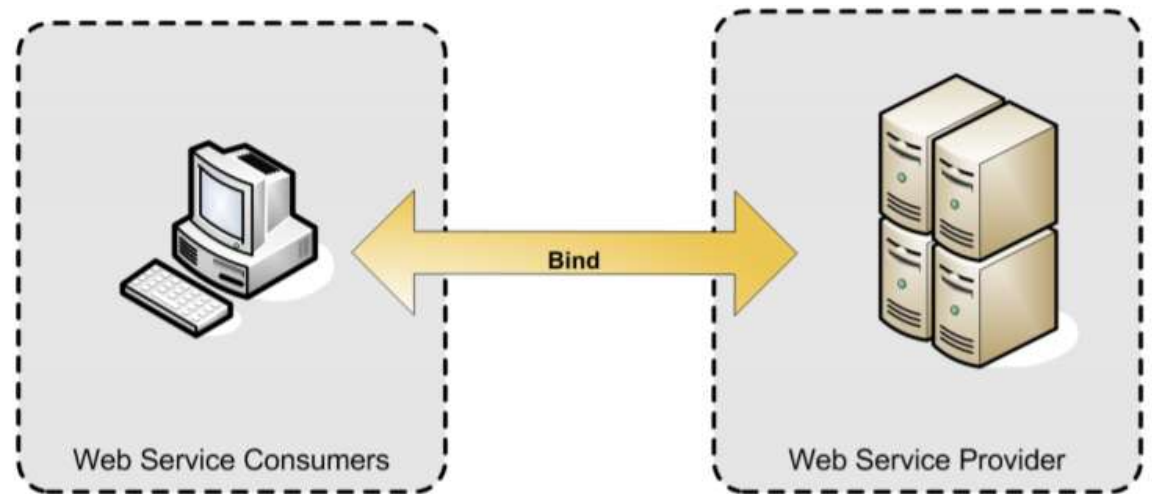
BENEFITS OF WEBSERVICES

- **Loosely Coupled**
 - Each service exists independently of the other services that make up the application.
 - Individual pieces of the application to be modified without impacting unrelated areas.
- **Ease of Integration**
 - Data is isolated between applications creating 'silos'.
 - Web Services act as glue between these and enable easier communications within and across organisations.
- **Service Reuse**
 - Takes code-reuse a step further and reduces duplication
 - (Theoretically) A specific function within the domain is only ever coded once and used over and over again by consuming applications.



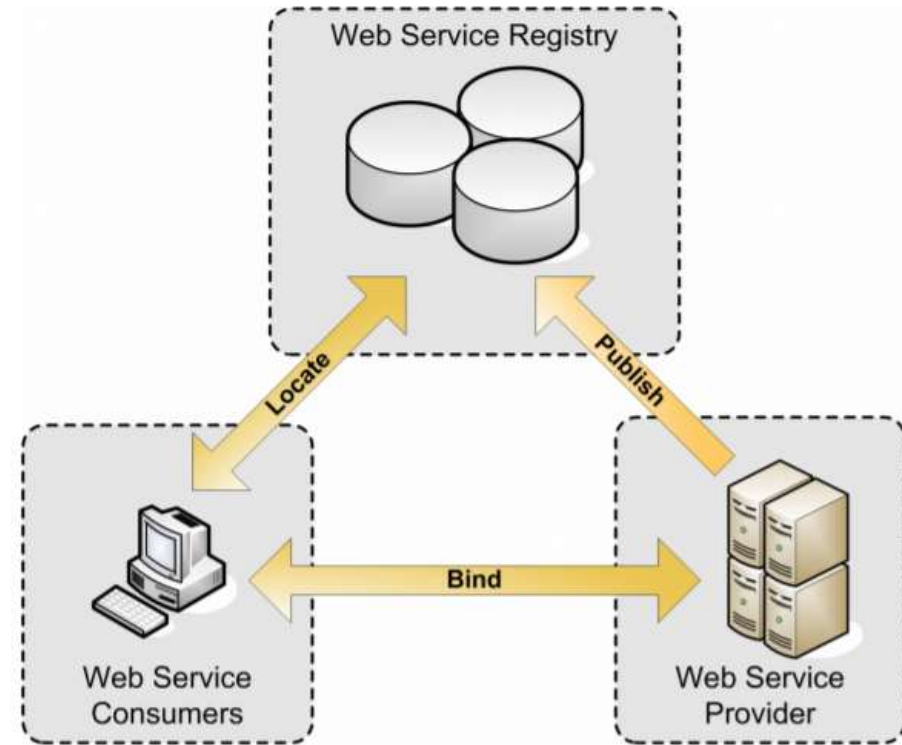
WEBSERVICES ARCHITECTURE (SIMPLE)

- The simplest Web service system has two participants:
 - A service producer (provider)
 - A service consumer (requester).
- The provider presents the interface and implementation of the service
- The requester uses the Web service.



WEBSERVICES ARCHITECTURE (COMPLEX)

- A registry, acts as a broker for Web services.
- A provider, can publish services to the registry
- A consumer, can then discover services in the registry and start using it.



WEBSERVICES (DECADE OLD DEFINITION)

- <https://www.w3.org/TR/ws-arch/#technology>

1.4 What is a Web service?

For the purpose of this Working Group and this architecture, and without prejudice toward other definitions, we will use the following definition:

[Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.]



WEBSERVICES

- DEFINITION (WSDL)
- DISCOVERY (UDDI)
- COMMUNICATION (SOAP over HTTP)
- All focuses on a vendor-neutral widely agreed upon message-exchange format



WHAT IS XML?

- XML is a software- and hardware-independent tool for storing and transporting data.
- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation



XML COMPONENTS

- **Elements**
 - The pairing of a start tag and an end tag.
- **Attributes**
 - A name-value pair that is part of a starting tag of an Element.
- **Processing Instructions**
 - Special directives to the application that will process the XML document.
- **Comments**
 - Messages helping a human reader understand the source code.
- **Character**
 - Data Characters (in a specific encoding) Entities Whitespace



Definition

The term **element** is a technical name for the pairing of a start tag and an end tag in an XML Document.

Production Rule

$$\begin{aligned}\langle element \rangle &::= \langle EmptyElement \rangle \\ &\quad | \langle STag \rangle \langle content \rangle \langle ETag \rangle \\ \langle STag \rangle &::= '<' \langle Name \rangle \langle Attribute \rangle^* '>' \\ \langle ETag \rangle &::= '</' Name '>' \\ \langle EmptyElement \rangle &::= '<' Name \langle Attribute \rangle^* '/>'\end{aligned}$$

- XML Elements must be strictly nested!
- Element names can include letters, the underscore, hyphen and colon; they **must** begin with a letter.
- Element names are case sensitive!



XML EXAMPLE

- show real XML
 - Elements
 - Processing tags
 - Etc

[W3Schools](#)

XML Example 1

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```



SOME PROBLEMS AND FIXES

- Physical Structure of the document
 - Well formedness (Parsers)
- Logical Structure of the document
 - Validity (Schemas). Semantics of the elements?
- Element Name clashes between Documents
 - Namespaces
- Lets see examples.....

XML Schema

XML Schema is an XML-based alternative to DTD:

```
<xs:element name="note">  
  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="to" type="xs:string"/>  
      <xs:element name="from" type="xs:string"/>  
      <xs:element name="heading" type="xs:string"/>  
      <xs:element name="body" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
  
</xs:element>
```



TUTORIAL EXERCISE — XML & SCHEMA

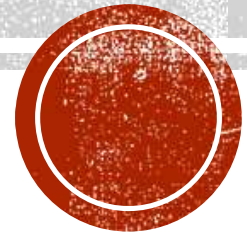
- Use the question assigned to you in FSD-2 Lab Exam
- Create a Schema for Storing the data as XML (XSD File)
- Create an XML document as an example and validate using the schema

- Turn in the following
 1. XSD file
 2. XML file
 3. Screenshot of validation



SOAP AND WSDL WEBSERVICES

The Classical!



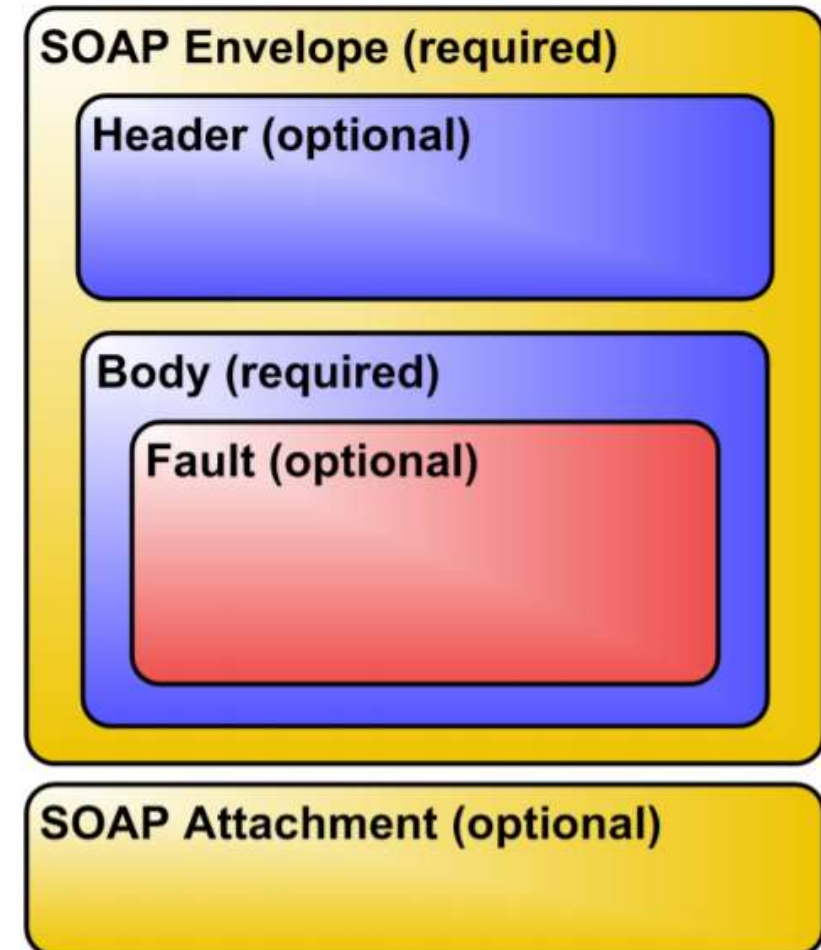
SIMPLE OBJECT ACCESS PROTOCOL

- ✦ SOAP is an industry accepted W3C specification for a ubiquitous XML distributed computing infrastructure.
 - ✦ A mechanism for defining the unit of communication.
 - ✦ A mechanism for error handling.
 - ✦ An extensibility mechanism Lives above the transport layer of OSI
 - ✦ Simply put its a mechanism that allows the transmission of XML documents, regardless of transport layer protocol.



STRUCTURE OF SOAP MESSAGES

- ✦ The root element of a SOAP message is the Envelope element.
- ✦ It contains an optional Header element and the required Body
- ✦ Elements called Faults can be used to describe exceptional situations.
- ✦ It can contain optional Attachments in MIME encoding for exchanging binary data.



SOAP MESSAGES

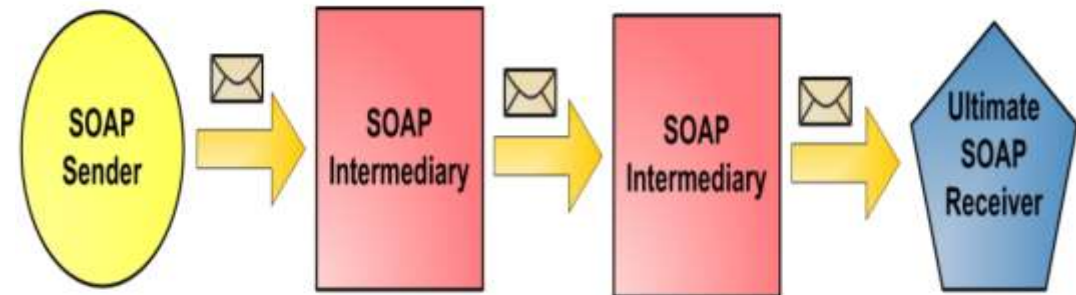
✦ Lets see some examples!

[W3Schools](#)



SOAP MESSAGE DELIVERY

- ✦ The SOAP Sender creates and sends a SOAP Message to an ultimate SOAP Receiver.
- ✦ One or more optional SOAP Intermediaries can be positioned to intercept messages between the the sender and the receiver. They can perform filtering, logging, catching etc.
- ✦ The SOAP sender's intended destination is called the Ultimate SOAP Receiver.



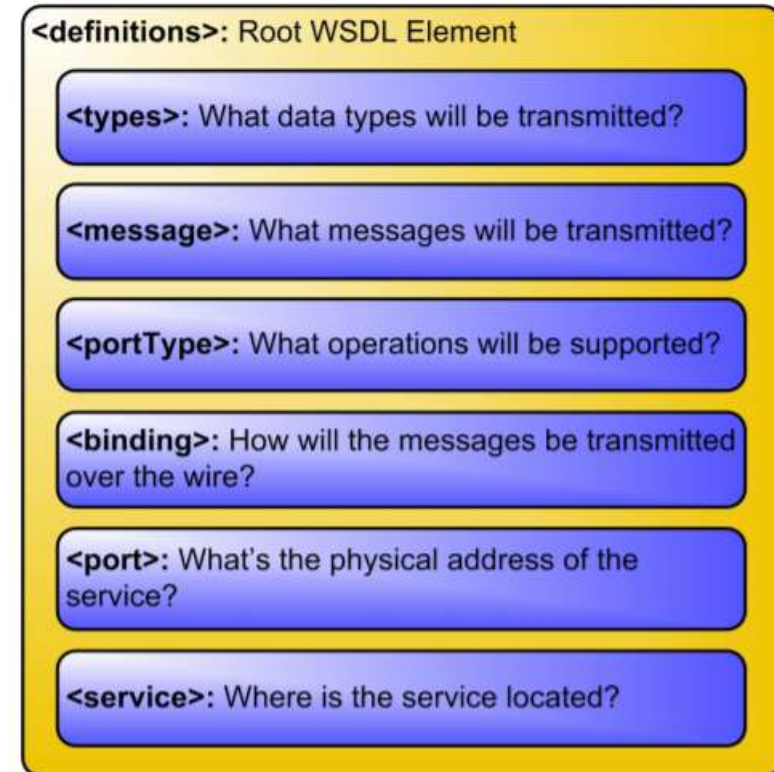
WEB SERVICES DEFINITION LANGUAGE

- ★ Web Services Description Language (WSDL) is an XML format for describing all the information needed to invoke and communicate with a Web Service.
- ★ It gives the answers to the questions Who? What? Where? Why? How?
- ★ A service description has two major components:
 - ★ Functional Description
 - ★ Defines details of how the Web Service is invoked, where it's invoked.
 - ★ Focuses on the details of the syntax of the message and how to configure the network protocols to deliver the message.
 - ★ Nonfunctional Description
 - ★ Provides other details that are secondary to the message (such as security policy) but instruct the requestor's runtime environment to include additional SOAP headers.



WSDL STRUCTURE

- ✦ A WSDL Document is a set of definitions with a single root element. Services can be defined using the following XML elements:
 - ✦ **Types**, think Data Type
 - ✦ **Message**, think Methods
 - ✦ **PortType**, think Interfaces
 - ✦ **Binding**, think Encoding Scheme
 - ✦ **Port**, think URL
 - ✦ **Service**, many URLs



WSDL EXAMPLE

✦ Let's see some real examples!

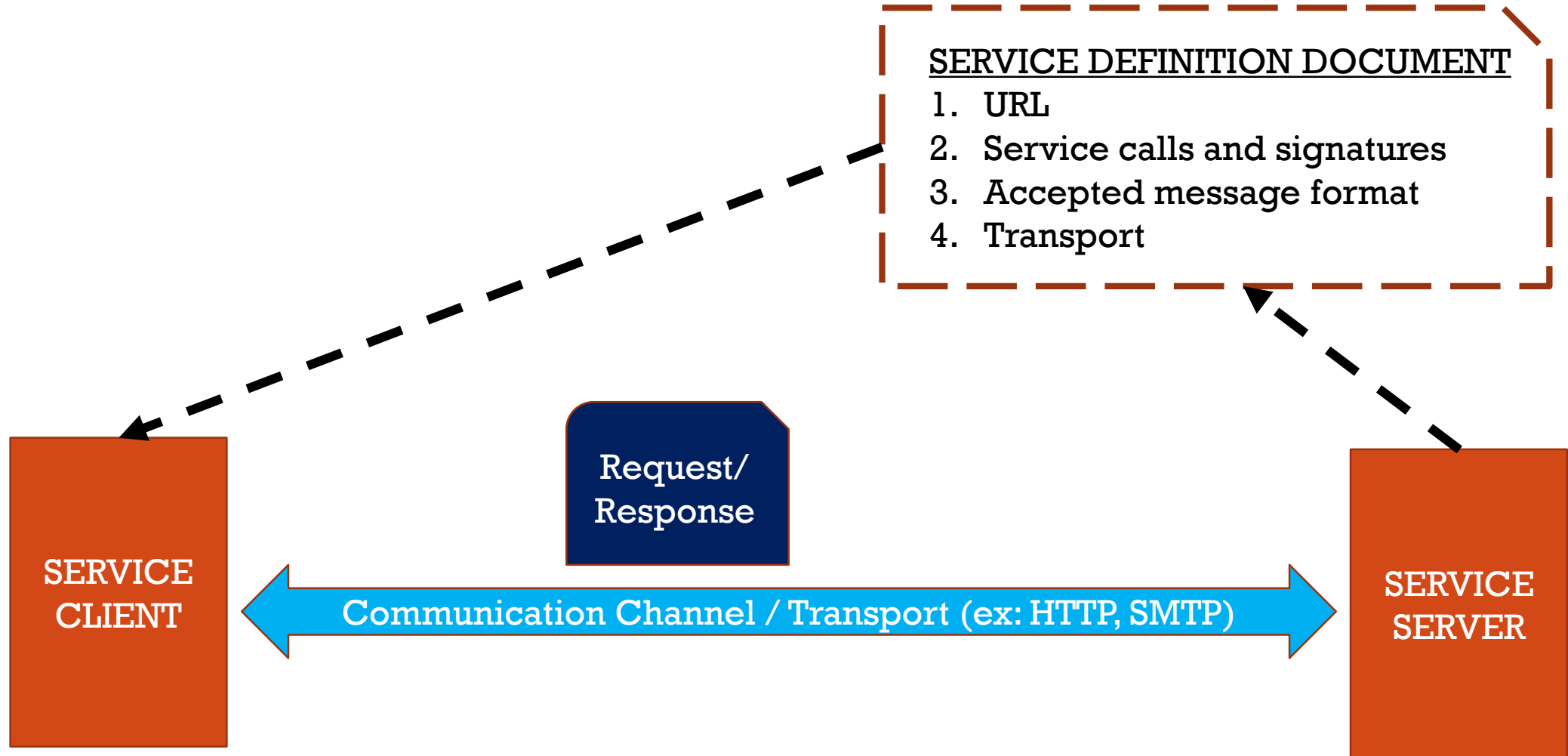
[W3Schools](#)



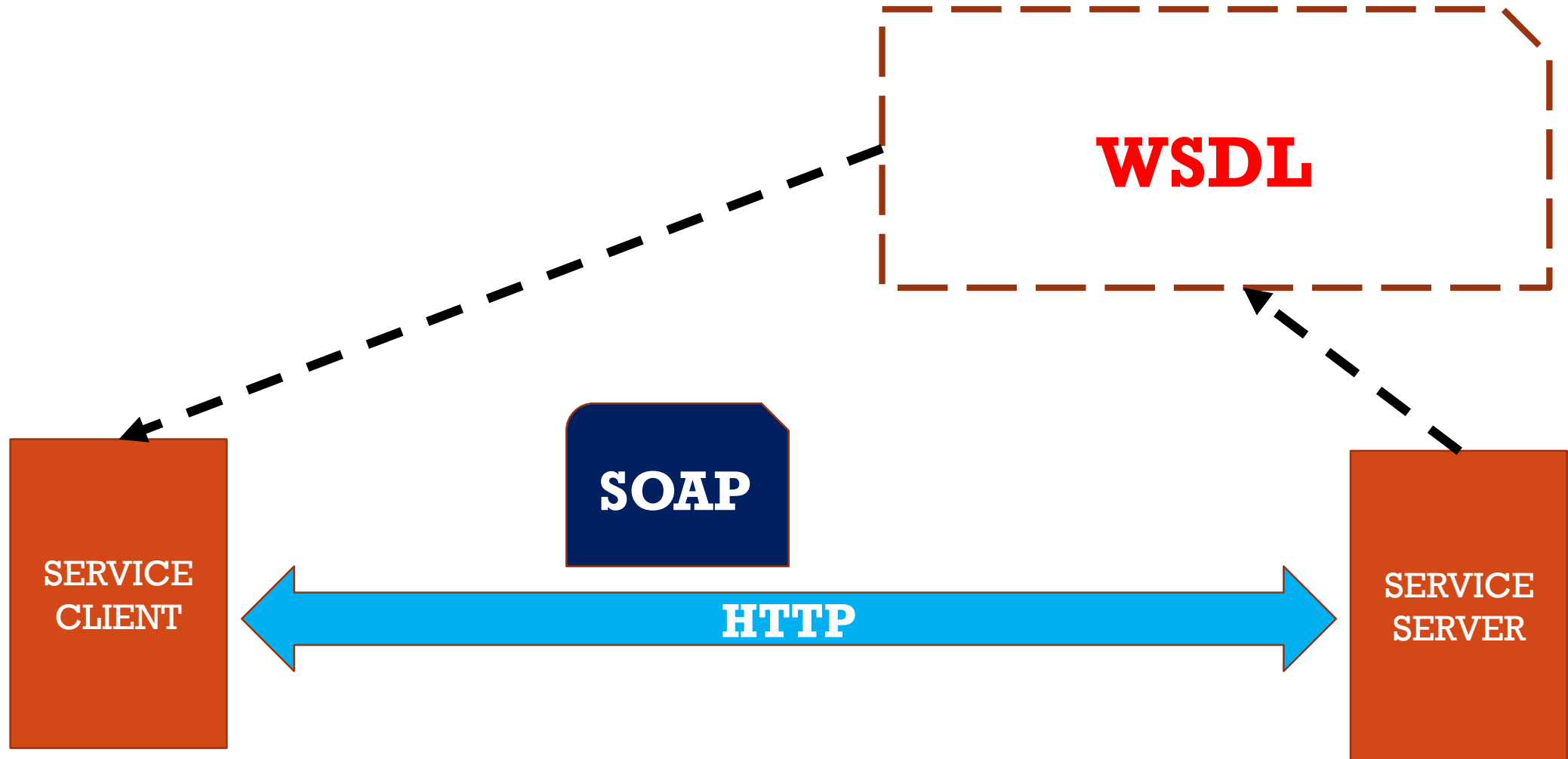
GENERAL ARCHITECTURE OF WEBSERVICES



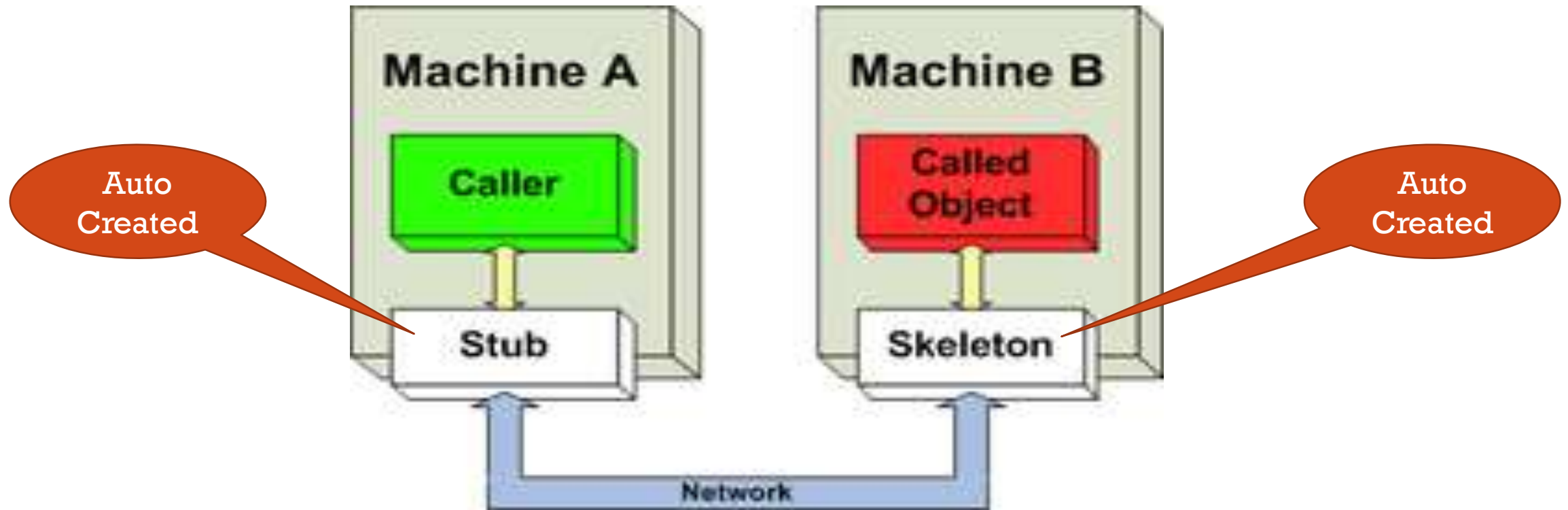
GENERAL ARCHITECTURE OF WEBSERVICES



TYPICAL WSDL SOAP WEBSERVICES



WSDL COMMON USAGE



LETS SEE A LIVE EXAMPLE!



PROBLEMS WITH WSDL-SOAP

- ✦ XML can become very verbose
 - ✦ So there is a tradeoff between bandwidth, data transfer, etc
- ✦ Primarily for heavy duty work
 - ✦ Can transfer binary data through attachments
- ✦ XML is designed for machine readability
 - ✦ Tedious to debug
 - ✦ Even though tools are available
- ✦ You don't need
 - ✦ A supercomputer for browsing facebook
 - ✦ A written contract and registrar for borrowing 10 Rs from a friend
- ✦ The necessity of webservises have become widespread
 - ✦ Unlike earlier we can imagine using webservises even for simpler things
 - ✦ The advent of smartphones have exacerbated this



REST TO THE RESCUE



- ✦ Piggybacks on HTTP verbs (we will soon see how)
- ✦ JSON format though not directly connected to REST, it goes hand-in-hand with the proliferation of RESTful services
- ✦ JSON is a pleasure to deal with (Thanks Doug Crockford)
 - ✦ (read) Javascript the good parts
- ✦ Cons
 - ✦ No machine readable definitions (at least till a while back)
 - ✦ Remember I am old!
 - ✦ Hence, No auto *stub* generation tools
 - ✦ But we don't need them anyway
 - ✦ because its simple

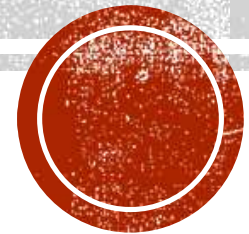


REFERENCE

- Slide Outlines thanks to :
<https://www.cl.cam.ac.uk/~ib249/teaching/Lecture1.handout.pdf>



WHY REST?



PROBLEMS WITH WSDL-SOAP

- ✦ XML can become very verbose
 - ✦ So there is a tradeoff between bandwidth, data transfer, etc
- ✦ Primarily for heavy duty work
 - ✦ Can transfer binary data through attachments
- ✦ XML is designed for machine readability
 - ✦ Tedious to debug
 - ✦ Even though tools are available
- ✦ You don't need
 - ✦ A supercomputer for browsing facebook
 - ✦ A written contract and registrar for borrowing 10 Rs from a friend
- ✦ The necessity of webservises have become widespread
 - ✦ Unlike earlier we can imagine using webservises even for simpler things
 - ✦ The advent of smartphones have exacerbated this



REST TO THE RESCUE

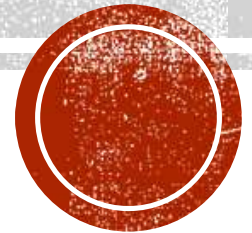


- ✦ Piggybacks on HTTP verbs (we will soon see how)
- ✦ JSON format though not directly connected to REST, it goes hand-in-hand with the proliferation of RESTful services
- ✦ JSON is a pleasure to deal with (Thanks Doug Crockford)
 - ✦ (read) Javascript the good parts
- ✦ Cons
 - ✦ No machine readable definitions (at least till a while back)
 - ✦ Remember I am old!
 - ✦ Hence, No auto *stub* generation tools
 - ✦ But we don't need them anyway
 - ✦ because its simple



REPRESENTATIONAL STATE TRANSFER (REST)

The theory of Representational State Transfer – The
Architectural Style – By Roy Fielding



UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

REST

PhD Dissertation of Roy Fielding (2000) at UCI. He was also the chairman of Apache Software foundation and a prolific contributor for WWW as member of W3C. He is one of the authors of **HTTP/1.1** specification. Currently Principal Scientist in Adobe



SOFTWARE DESIGN VS NETWORKING RESEARCH

- Software Design research
 - the categorization of software designs and the development of design methodologies
 - but **NOT** the impact of various **design choices on system behavior**
- Networking research
 - Generic communication behavior between systems
 - Improving the performance of particular communication techniques
 - But **NOT** the fact that changing the **interaction style of an application** can have more impact on performance than the communication protocols.



FIELDING'S FOCUS

“ My work is motivated by the desire to understand and evaluate the **architectural design** of **network-based application software** through principled use of **architectural constraints**, thereby obtaining the ...properties desired of an architecture. ”



SOFTWARE ARCHITECTURE

“A software architecture is an **abstraction** of the **run-time elements** of a software system during some **phase** of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.”



SOFTWARE ARCHITECTURE - ABSTRACTION

- At the heart of software architecture is the principle of abstraction
 - A complex system will contain many levels of abstraction, each with its own architecture.
- Architectural elements are delineated by the abstract interfaces
 - Within each element may be found another architecture
- The *sub*-architecture
 - Defines the system of sub-elements that implement the behavior represented by the parent element's abstract interface



SOFTWARE ARCHITECTURE - PHASES

- In addition, a software system will often have multiple operational phases
 - start-up
 - Initialization
 - normal processing
 - re-initialization
 - shutdown.
- Each operational phase has its own architecture
- For example, a configuration file will be treated as a data element during the start-up phase, but won't be considered otherwise
- An overall description of a system architecture should also describe the architecture of transitions between phases



SOFTWARE ARCHITECTURE - ELEMENTS

- Perry and Wolf [105] define processing elements as “transformers of data”
- Shaw et al. [118] describe components as “the locus of computation and state.”

“A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters.”

“A software architecture is defined by a configuration of architectural elements: **components, connectors, and data**—constrained in their relationships in order to achieve a desired set of architectural properties.”

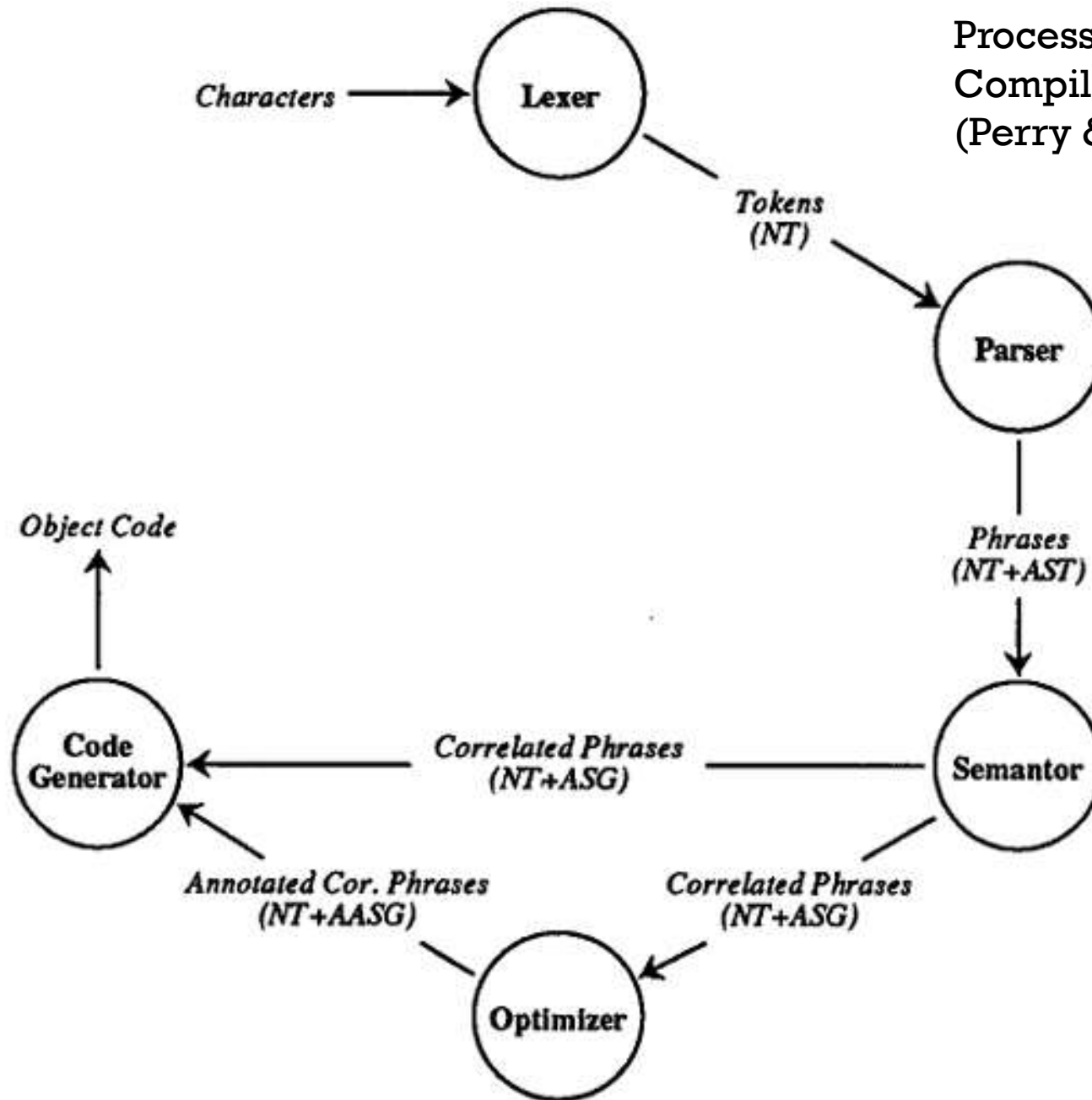


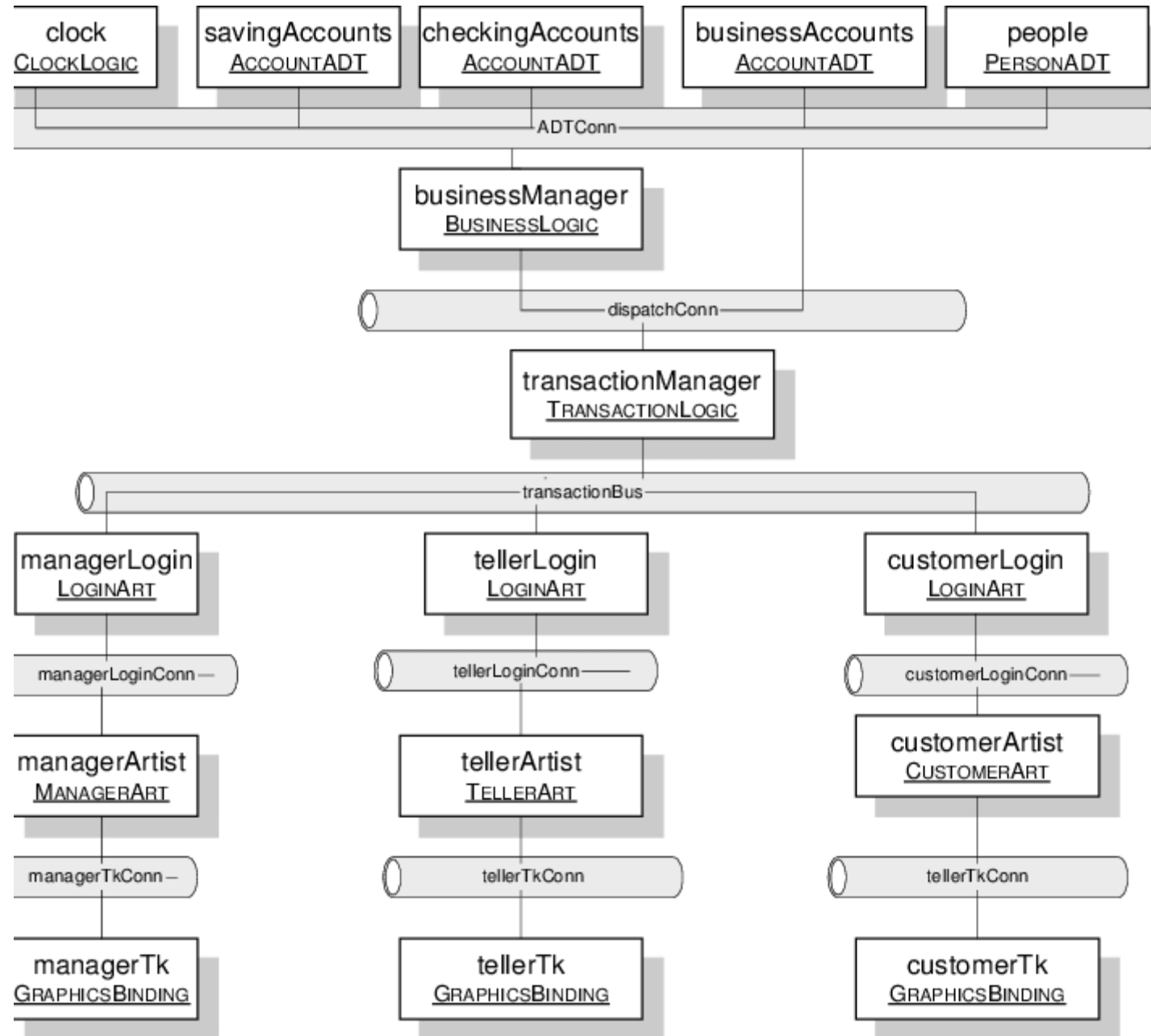
COMPONENTS, CONNECTORS & DATA

1. A **component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.
2. A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.
3. A **datum** is an element of information that is transferred from a component, or received by a component, via a connector.



Processing View of Sequential
Compiler Architecture
(Perry & Wolf 1992)





SOFTWARE ARCHITECTURE VS SOFTWARE STRUCTURE

- There is an important distinction between software architecture and software structure
- **Software Architecture** is an abstraction of the run-time behavior of a software system.
- **Software Structure** on the other hand is a property of the static software source code.
- Sometimes the structure may correspond well with the architecture.
 - But, not always.
 - Also, this is not a necessary condition



LET'S DISCUSS

- Describe an NLP Software (assumed)



ARCHITECTURAL PROPERTIES

- The set of architectural properties includes all properties that derive from the selection and arrangement of components, connectors, and data.
- Functional properties (Application behavior specification)
- Non-Functional properties (ease of use, efficiency, etc)
- Properties are achieved via architectural constraints (we will see next slide)
- The goal of **architectural design** is to create an architecture with a set of architectural properties that form a **superset** of the system (functional & non-functional) requirements.



ARCHITECTURAL CONSTRAINTS

- Constraints are often motivated by Software design principles/patterns.
- Remember these.... constraints
 1. The model is **only responsible** for managing the data of the application. It receives user input from the controller.
 2. The view is **only responsible** for rendering presentation of the model in a particular format.
 3. The controller **is only responsible** for responds to the user input and performs interactions on the data model objects.
- If you follow all these constraints, then it can be said that you are adhering to **Model-View-Controller** Design pattern.



ARCHITECTURAL STYLE

- There is no authority. Another name may be “Architectural pattern”

“An architectural style is a **coordinated set of architectural constraints** that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.”



NETWORK BASED VS DISTRIBUTED

- Distributed system is one that looks to its users like an ordinary centralized system, but runs on multiple, independent CPUs.
- Network-based systems are those capable of operation across a network
 - but not necessarily transparent to the user.
 - It may even be desirable for user to be aware of the difference between an action that requires a network request
- The focus of Roy Fielding is about **Network Based Application Architectures**



NETWORK BASED ARCHITECTURE- DATA FLOW STYLES

- Pipe and Filter style (PF)
 - Each component (filter) reads streams of data on its inputs and produces streams of data on its outputs
 - Usually while applying a transformation to the input streams and processing them incrementally
 - So output begins before the input is completely consumed
- Uniform Pipe and Filter style (UPF)
 - Adds the constraint that all filters must have the same interface.
 - Example: Unix (ish) operating system styles



EXAMPLE OF STYLE INDUCED PROPERTIES (PF & UPF)


- PF allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of the individual filters (**simplicity**).
- PF supports reuse: any two filters can be hooked together (**reusability**)
- PF systems can be easily maintained and enhanced: new filters can be added to existing systems (**extensibility**)
- In UPF, Restricting the interface allows independently developed filters to be arranged at will to form new applications

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	-	±			++	+	+		++	++	+		

Table 3-1. Evaluation of Data-flow Styles for Network-based Hypermedia



NETWORK BASED ARCHITECTURE- REPLICATION STYLES

- Replicated Repository Style (RRS)
 - Improve the accessibility of data and scalability of services by having more than one process provide the same service
 - Provide clients the illusion that there is just one centralized service
- Cache Style (\$) 
 - A variant of replicated
 - Replication of the result (response) of an individual request such that it may be reused by later requests

NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Client Server Style (CS)

- A server component, offering a set of services, listens for requests upon those services.
- A client component, desiring that a service be performed, sends a request to the server via a connector.

- Layered Client Server Style (LCS)

- Each layer providing services to the layer above it and using services of the layer below it
- Reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer
- Thus improving evolvability and reusability.
- Layered-client-server adds proxy and gateway components to the client-server style.



NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Client Stateless Server Style (**CSS**)
 - Derives from client-server with the additional constraint that no session state is allowed on the server component.
 - Each request from client to server must contain all of the information necessary to understand the request.
 - Session state is kept entirely on the client.
- Client-Cache-Stateless-Server (**C\$SS**)
 - Derives from the CSS and adds the \$ style
 - A cache acts as a mediator between client and server
 - The responses (if cacheable) can be reused



NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Layered-Client-Cache-Stateless-Server (LC\$CSS)
 - Derives from both LCS and C\$SS through the addition of proxy and/or gateway components.
- As you can see, when styles have **non-conflicting set of constraints** they can be stacked up to create new styles



NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Virtual Machine (**VM**)
 - The code must be executed in some fashion, preferably within a controlled environment to satisfy security and reliability concerns
- Remote Evaluation (**REV**)
 - Derived from the client-server (CS) and virtual machine (VM) styles
 - A client component has the know-how necessary to perform a service
 - But client lacks the resources (CPU cycles, data source, etc.) required
 - The client sends the know-how to a server component at the remote site
 - The server executes the code using the resources available



NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Code on Demand (COD)
 - A client component has access to a set of resources, but not the know-how
 - The client sends a request to a remote server for the code representing that know-how
 - The server sends it and client receives that code, and executes it locally



NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Layered-Code-on-Demand-Client-Cache-Stateless-Server (**LCODC\$SS**)
 - Derived by incorporating code-on-demand to the layered-client-cache-stateless-server (LC\$SS)
 - Example Java Applets (extinct – don't use)



NETWORK BASED ARCHITECTURE- OTHER STYLES

- Other Hierarchical Styles
 - Remote session (example: cookies)
 - Remote Data Access (example: JDBC)
- Other Mobile Code Styles
 - Mobile Agent
- Peer to Peer Style
 - Event Based Integration
 - C2 Architectural Style
- Blackboard architectural style
- Many more....But we will focus on the ones that are important for ***deriving*** the REST framework



REPRESENTATIONAL STATE TRANSFER (REST)

- REST is an **architectural style** for **Distributed Hypermedia System**
- REST is a hybrid style **derived** from several of the network-based architectural styles seen so far
- With one additional (Most) important constraint: **Uniform Connector Interface**



DERIVING REST

- The first constraints added to our hybrid style are those of the client-server architectural style (CS)
 - Separation of concerns is the principle behind the client-server
 - Separation allows the components to evolve independently
 - Separating UI responsibility leads to improved portability (multiplatform)
- Next we improve on this to Client Stateless Server style (CSS)
 - Each request should contain all the server needs
 - Session state should be maintained entirely on the client
 - This constraint induces the properties of visibility, reliability, and scalability.



DERIVING REST

- Next we improve on this to Client Cache Stateless Server style (C\$SS)
 - If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
 - This improves efficiency, scalability, and user-perceived performance
- Next we improve on this to LC\$SS
 - Layered system style allows an architecture to be composed of hierarchical layers
 - Each component cannot “see” beyond the immediate layer
 - This bounds the overall system complexity and promotes substrate independence.
 - Layers can be used to encapsulate legacy services and to protect new services from legacy clients
- The central feature that distinguishes REST is the Uniform Interface style, which we will discuss separately in the next slide



DERIVING REST

- Next we add Code on Demand Style to derive (LCODC\$CSS)
 - This allows remote code distribution.
 - Remember REST is a general architectural style applicable for Networked Hypermedia application software.
 - We are only interested in REST style as applied to API/Services design, so we will not focus more on this.
- Finally, the central feature that distinguishes REST is the Uniform Interface style, which we will discuss separately in the next slide



UNIFORM INTERFACE STYLE

- Emphasizes a uniform interface between components
- The overall system architecture is simplified and the visibility of interactions is improved
- Trade-off: degrades efficiency
 - Information is transferred in a standardized form rather than one which is specific to the application's needs.
- In order to get a Uniform interface, this is further decomposed into four constraints
 - REST interface constraints



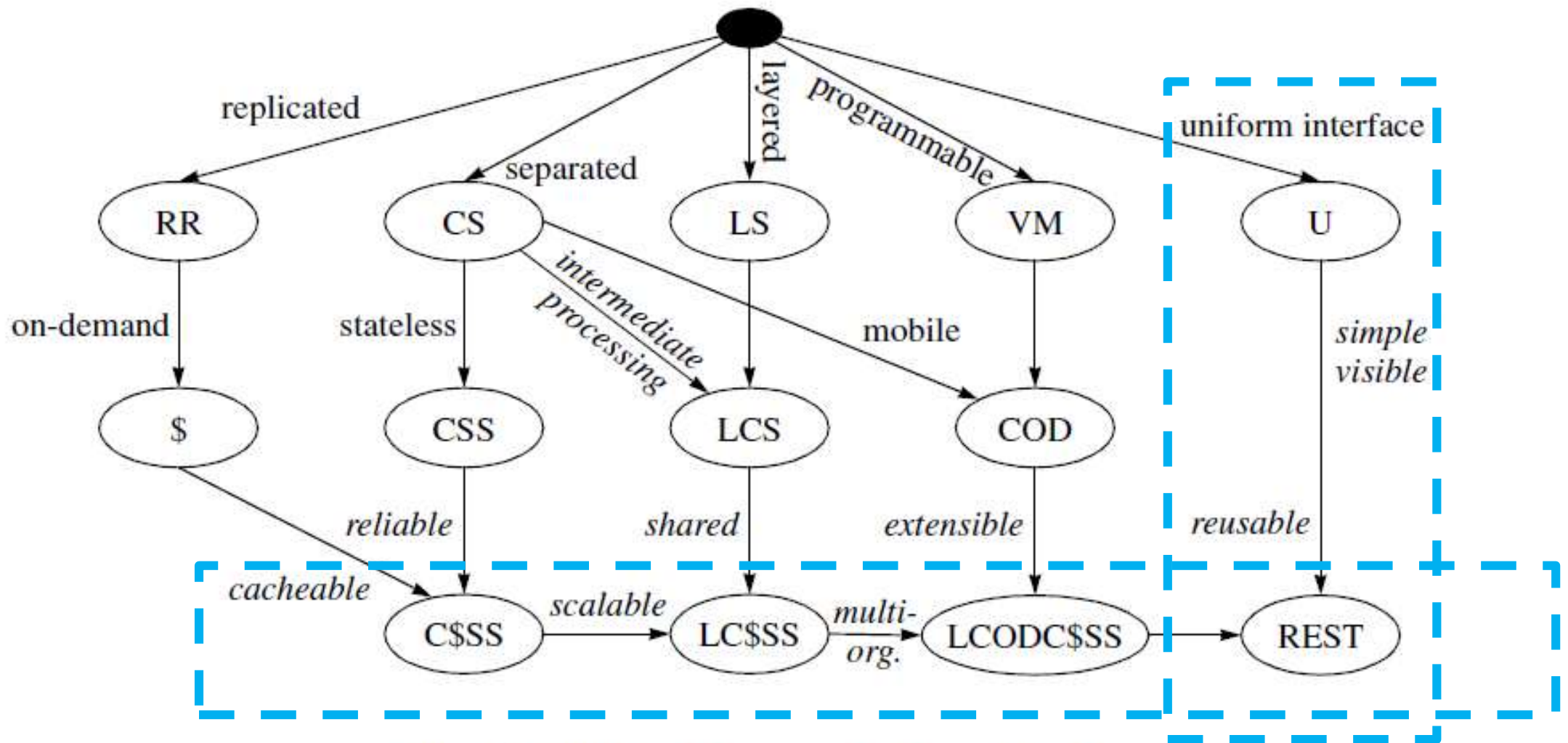


Figure 5-9. REST Derivation by Style Constraints



REST INTERFACE CONSTRAINTS

1. Identification of resources
2. Manipulation of resources through representations
3. Self-descriptive messages
4. Hypermedia as the engine of application state



REST RESOURCE IDENTIFICATION

- The key abstraction of information in REST is a **resource**.
- Any information that can be named can be a resource:
 - a document or image,
 - a temporal service (e.g. “today’s weather in Los Angeles”)
- REST uses a resource identifier to identify the particular resource involved in an interaction between components.
- REST relies on the author choosing a resource identifier that best fits the nature of the concept being identified



RESOURCE REPRESENTATION

- REST components perform actions on a resource by using a **representation** to capture the current or intended state of that resource and transferring that representation between components.
- A representation is a sequence of bytes, plus representation metadata to describe those bytes
 - Document, file, and HTTP message entity, etc
- The data format of a representation is known as a *media type*
- The resource is a conceptual mapping
 - the server receives the identifier (which identifies the mapping) and applies it to its current mapping implementation (usually a combination of collection-specific deep tree traversal and/or hash tables)
 - To find the currently responsible handler implementation
 - The handler implementation then selects the appropriate action+response based on the request content.



SELF DESCRIPTIVE MESSAGES

- Component interactions occur in the form of dynamically sized messages
- The most frequent form of request semantics is that of retrieving a representation of a resource (e.g., the “GET” method in HTTP)
 - which can often be cached for later reuse
- REST eliminating any need for the server to maintain an awareness of the client state beyond the current request



“**Representational State Transfer**” is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (**state transitions**), resulting in the next page (representing the next state of the application) being **transferred** to the user and rendered for their use..



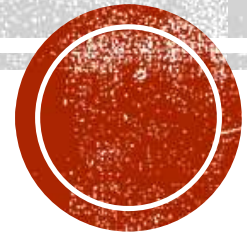
NOTES ON REST DEVELOPMENT

- The first edition of REST was developed between October 1994 and August 1995
- REST was the means for communicating Web concepts as during HTTP/1.0 specification and the initial HTTP/1.1 proposal.
- REST was iteratively improved over the next five years and applied to various revisions and extensions of the Web protocol standards.
- REST was originally referred to as the “HTTP object model,” but that name would often lead to misinterpretation of it as the implementation model of an HTTP server.



RESTFUL SERVICES

Richardson Maturity Model – *Martin Fowler*



Glory of REST



Level 3: Hypermedia Controls

Level 2: HTTP Verbs

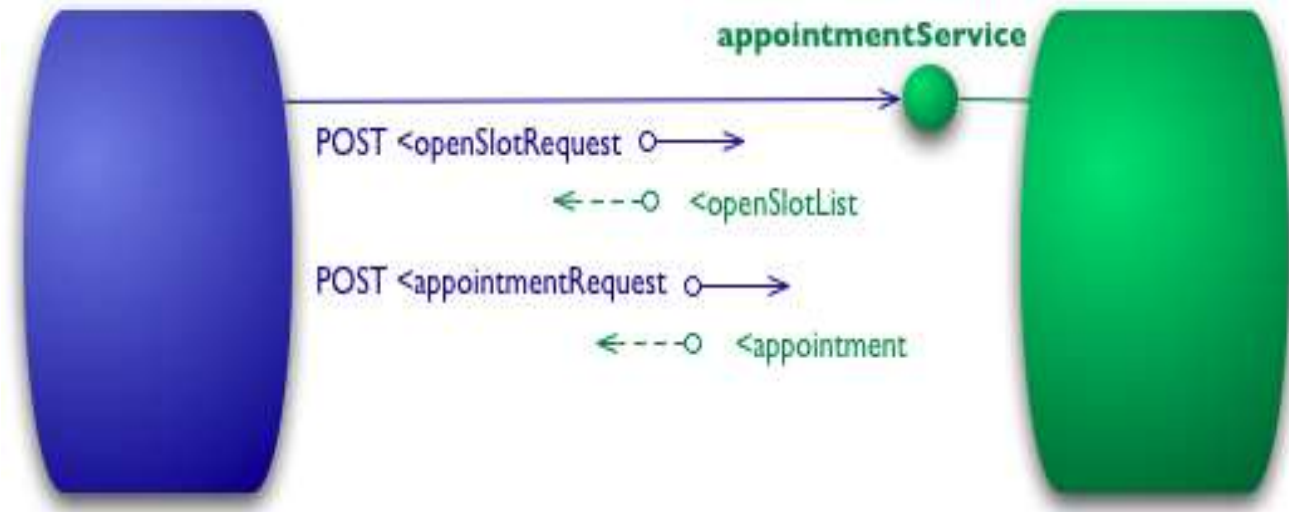
Level 1: Resources

Level 0: The Swamp of POX



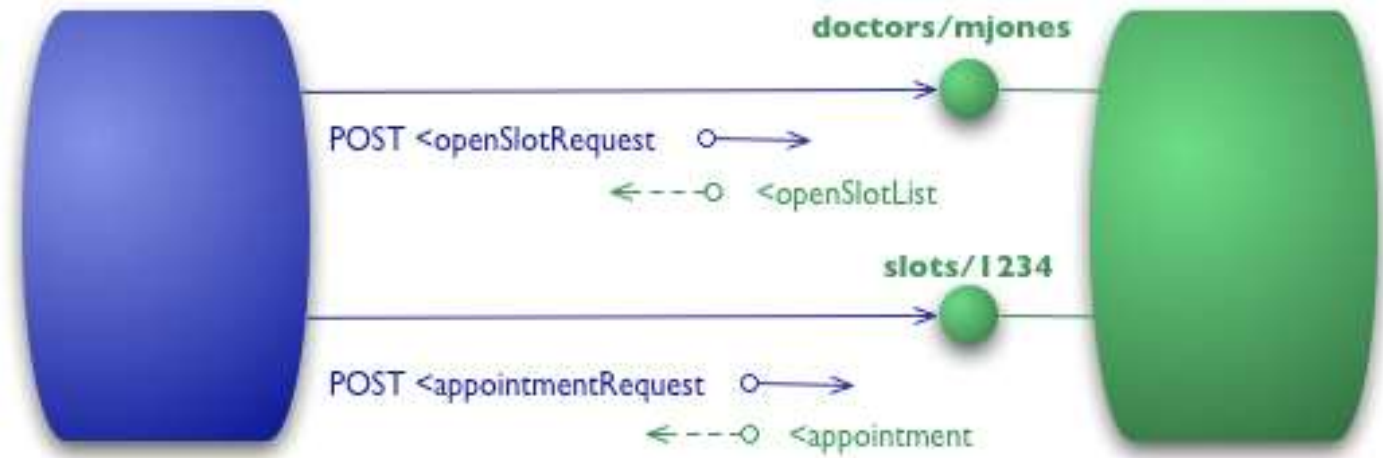
LEVEL 0

- HTTP as a transport system for remote interactions
- HTTP as a tunneling mechanism for your own Remote Procedure Invocation
- the content can actually be anything: JSON, YAML, key-value pairs, or any custom format



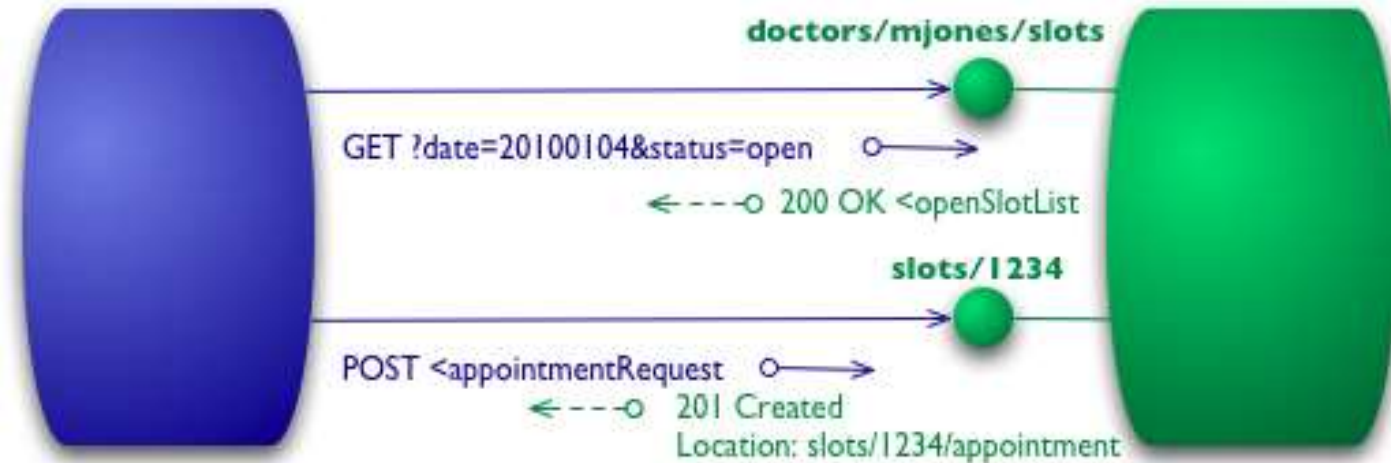
LEVEL 1

- The next step is the introduction of “resources”
- Instead of endpoints, we start communicating with resources
- Very similar to Object Oriented Programming Concepts



LEVEL 2

- Using the HTTP verbs as closely as possible to how they are used in HTTP itself
- Interpretation
 - GET (collection)
 - GET/id (individual)
 - POST (creation)
 - PATCH/id (update individual)
 - DELETE/id (delete individual)

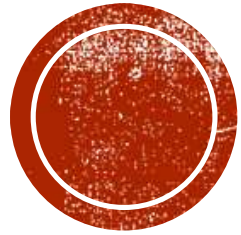


LEVEL 3

- **HATEOAS** (Hypertext As The Engine Of Application State)
- Hypermedia controls is that they tell us what we can do next, and the URI of the resource we need to manipulate
- One obvious benefit of hypermedia controls is that it allows the server to change its URI scheme without breaking clients
- There's no absolute standard as to how to represent hypermedia controls. The 'REST in Practice' team, recommends ATOM (RFC 4287)

```
1 <appointment>
2   <slot id = "1234" doctor = "mjones" start = "1400" end
   = "1450"/>
3   <patient id = "jsmith"/>
4   <link rel = "/linkrels/appointment/cancel"
5     uri = "/slots/1234/appointment"/>
6   <link rel = "/linkrels/appointment/addTest"
7     uri = "/slots/1234/appointment/tests"/>
8   <link rel = "self"
9     uri = "/slots/1234/appointment"/>
10  <link rel = "/linkrels/appointment/changeTime"
11    uri = "/doctors/mjones/slots?date=20100104&status
    =open"/>
12  <link rel = "/linkrels/appointment/updateContactInfo"
13    uri = "/patients/jsmith/contactInfo"/>
14  <link rel = "/linkrels/help"
15    uri = "/help/appointment"/>
16 </appointment>
17 |
```





RESTFUL SERVICES



The Driver's view

“Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web”

- IBM

“ A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding. ”

- RedHat



GUIDELINES FOR RESTFUL SERVICE

1. Think **Resources** not End Points
 - There should be a way to uniquely identify each resource. URI is the de facto standard.
 - Has similarities to Object Oriented Design
2. **Uniform Interface**
 - Piggy back on HTTP Verbs (GET, POST, PATCH & DELETE) and also HTTP Response Codes
3. Decoupled **Representations**
 - Content can be accessed in variety of formats (json preferred for webservices)
4. **HATEOAS** - Hypermedia as the Engine of Application State
 - Hyperlink driven, explicit state transfer, stateless otherwise



GUIDELINES (EXTENDED)

1. Client- Server Architecture

- RESTful style is mostly applicable (only) for 'web'-services

2. Stateless Communication

- No client information is stored between requests and each request is separate and unconnected

3. Cacheable Data

- Cache wherever applicable to cutdown on roundtrips

Note: Remember that there is no authority for styles. Once you are developer, prepare yourself for heated water cooler debates about what should and should not be included in the RESTful style..



GUIDELINES (EXTENDED)

1. Client- Server Architecture

- RESTful style is mostly applicable (only) for 'web'-services

2. Stateless Communication

- No client information is stored between requests and each request is separate and unconnected

3. Cacheable Data

- Cache wherever applicable to cutdown on roundtrips

Note: Remember that there is no authority for styles. Once you are developer, prepare yourself for heated water cooler debates about what should and should not be included in the RESTful style..



HTTP METHODS/VERBS

HTTP Method	CRUD	Collection Resource (e.g. /users)	Single Resource (e.g. /users/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID	Avoid using POST on a single resource
GET	Read	200 (OK), list of users. Use pagination, sorting, and filtering to navigate big lists	200 (OK), single user. 404 (Not Found), if ID not found or invalid
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid



HTTP METHODS/VERBS

HTTP Method	CRUD	Collection Resource (e.g. /users)	Single Resource (e.g. /users/123)
PATCH	Partial Update/Modify	405 (Method not allowed), unless you want to modify the collection itself	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution	200 (OK). 404 (Not Found), if ID not found or invalid



HTTP RESPONSE CODES

HTTP defines standard status codes that can be used to convey the results of a client's request. The status codes are divided into five categories.

1. 1xx: Informational – Communicates transfer protocol-level information.
2. 2xx: Success – Indicates that the client's request was accepted successfully.
3. 3xx: Redirection – Indicates that the client must take some additional action in order to complete their request.
4. 4xx: Client Error – This category of error status codes points the finger at clients.
5. 5xx: Server Error – The server takes responsibility for these error status codes.

<https://restfulapi.net/http-status-codes/>



HTTP RESPONSE CODES — IMPORTANT ONES

- **200 OK** Indicates that the request has succeeded.
- **201 Created** Indicates that the request has succeeded and a new resource has been created as a result.
- **400 Bad Request** The request could not be understood by the server due to incorrect syntax. The client SHOULD NOT repeat the request without modifications.
- **401 Unauthorized** Indicates that the request requires user authentication information. The client MAY repeat the request with a suitable Authorization header field
- **404 Not Found** The server can not find the requested resource.
- **500 Internal Server Error** The server encountered an unexpected condition that prevented it from fulfilling the request.



Customer

Primary Key

Cust ID	Cust Name	Shipping Address	Newsletter
at_smith	Alan Smith	35 Palm St, Miami	Xbox News
roger25	Roger Banks	47 Campus Rd, Boston	PlayStation News
wilson44	Evan Wilson	28 Rock Av, Denver	Xbox News
wilson44	Evan Wilson	28 Rock Av, Denver	PlayStation News
am_smith	Alan Smith	47 Campus Rd, Boston	PlayStation News

Products

Primary Key

Item	Supplier	Supplier Phone	Price
Xbox One	Microsc	(800) BUY-XBOX	250
PlayStation 4	Sony	(800) BUY-SONY	300
PS Vita	Sony	(800) BUY-SONY	200

Join

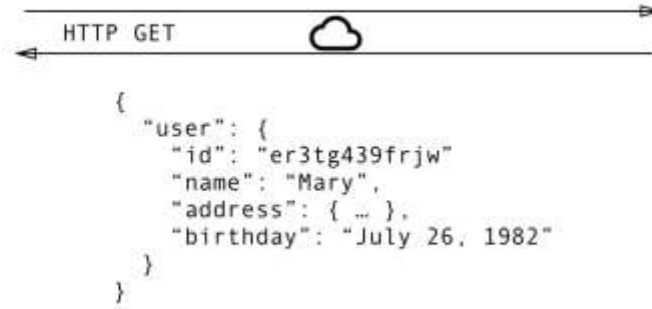
Primary Key

Primary Key

Cust ID	Item
at_smith	Xbox One
roger25	PlayStation 4
wilson44	Xbox One
wilson44	PS Vita
am_smith	PlayStation 4



1



/users/<id>

/users/<id>/posts

/users/<id>/followers



2



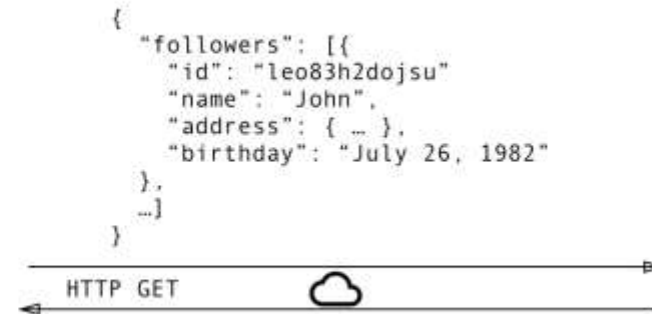
/users/<id>

/users/<id>/posts

/users/<id>/followers



3



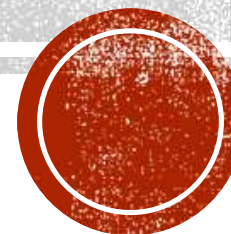
/users/<id>

/users/<id>/posts

/users/<id>/followers



GRAPHQL



REST HAS SOME PROBLEMS

- Over Fetching
- Under Fetching



WHY THESE 'PROBLEMS' MATTER

- Increased mobile clients (smart phones and smart devices) requires efficient data loading
- Client Heterogeneity (Example: Admin client vs Order Page -> accessing list of customers-orders)
- Faster development and deployment and rapid feature updates (hmm....grain of salt)



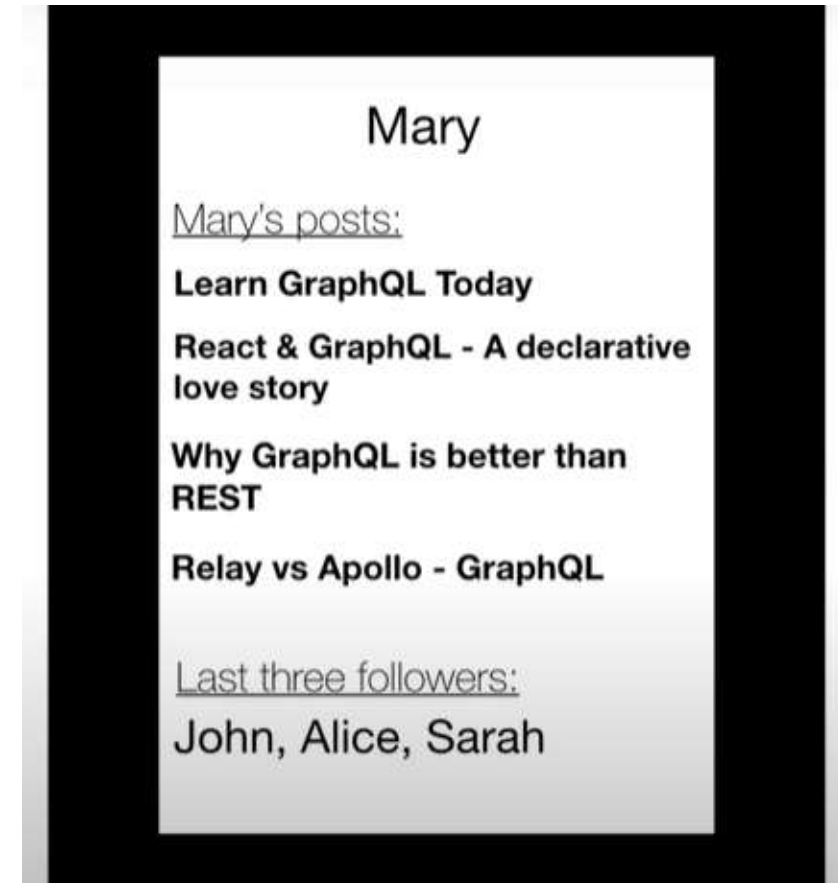
A LITTLE BIT OF HISTORY

- A Facebook (Meta) contribution
- Facebook started using it internally since 2012
- Made it public in 2015 (in a React.js Conference) and open sourced
- Lot of major companies have migrated their endpoints (or created new ones) to support GraphQL



GRAPHQL VS REST (BLOG APP)

- Print User Information
- List of Posts
- List of latest 3 followers
- Lets assume that the data is normalized
 - i.e. there are separate tables for
 1. User
 2. Posts
 3. Followers



GRAPHQL VS REST (BLOG APP)

- First Get The User Information

1



```
{  
  "user": {  
    "id": "er3tg439frjw"  
    "name": "Mary",  
    "address": { ... },  
    "birthday": "July 26, 1982"  
  }  
}
```

/users/<id>

/users/<id>/posts

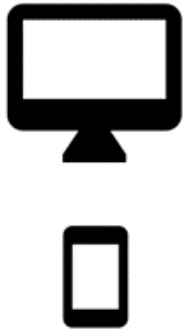
/users/<id>/followers



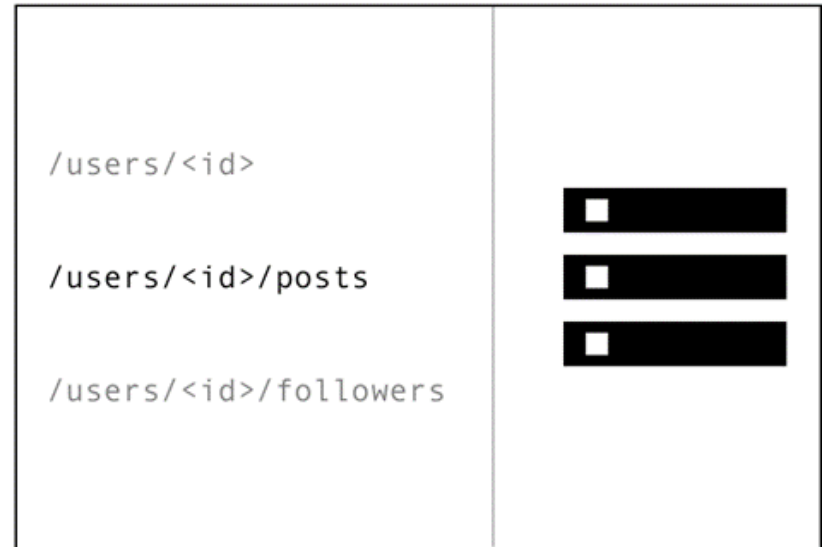
GRAPHQL VS REST (BLOG APP)

- Second Get The Posts Information

2



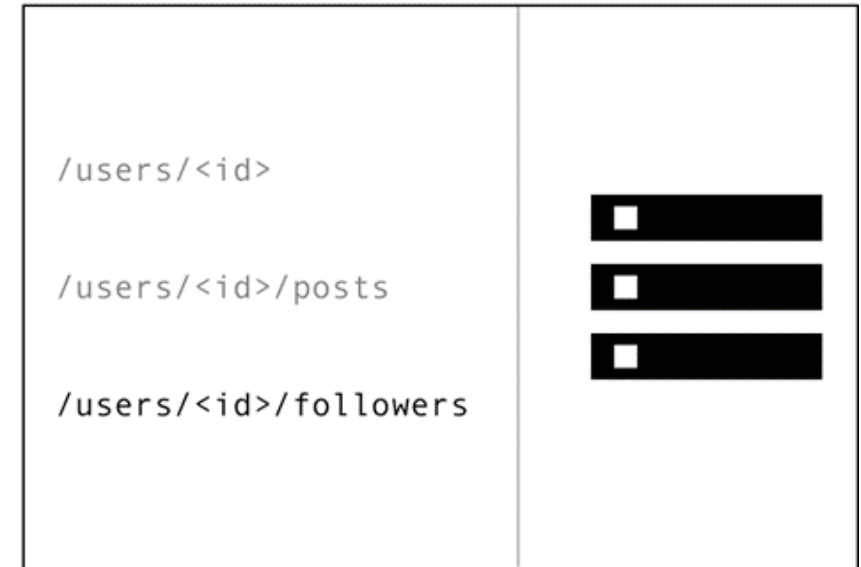
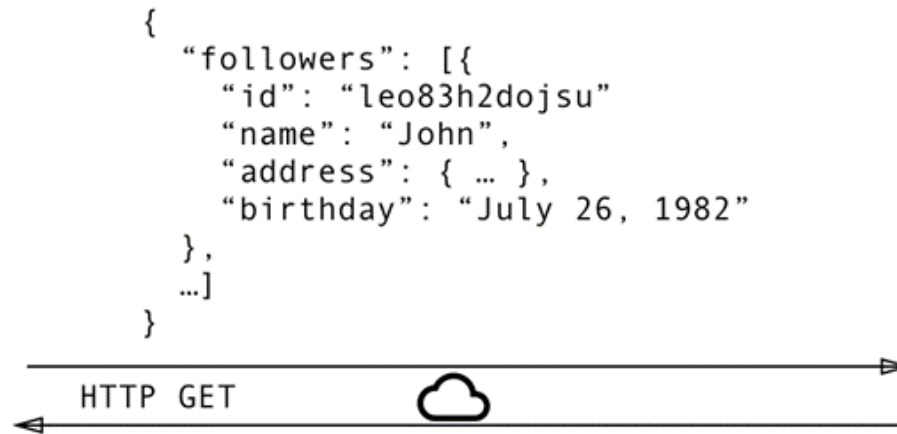
```
{  
  "posts": [{  
    "id": "ncwon3ce89hs"  
    "title": "Learn GraphQL today",  
    "content": "Lorem ipsum ...",  
    "comments": [ ... ],  
  }]  
}
```



GRAPHQL VS REST (BLOG APP)

- Finally Get The Followers Information

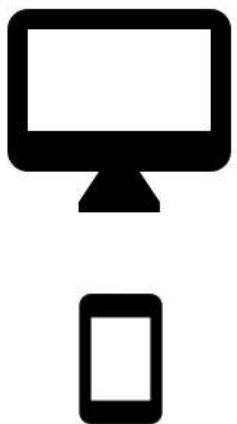
3



HOW DOES THE SAME THING LOOKS IN GRAPHQL?

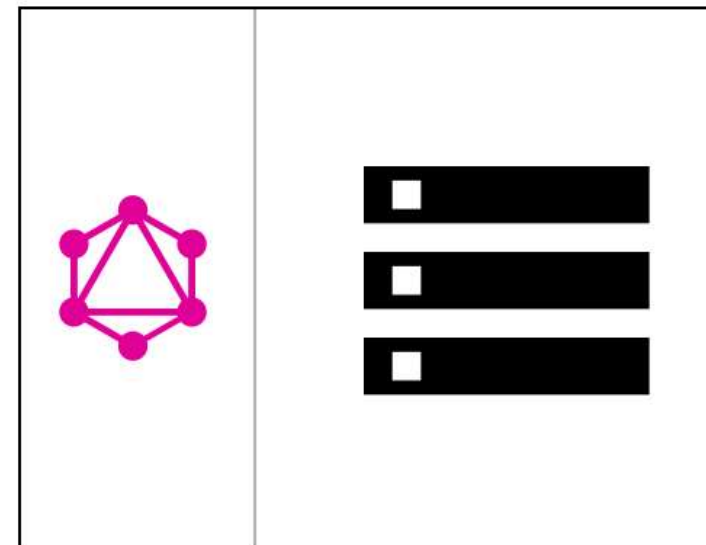
- Lets see!





```
query {  
  User(id: "er3tg439frjw") {  
    name  
    posts {  
      title  
    }  
    followers(last: 3) {  
      name  
    }  
  }  
}
```

Specify Only
What you
need



```
{  
  "data": {  
    "User": {  
      "name": "Mary",  
      "posts": [  
        { title: "Learn GraphQL today" }  
      ],  
      "followers": [  
        { name: "John" },  
        { name: "Alice" },  
        { name: "Sarah" },  
      ]  
    }  
  }  
}
```

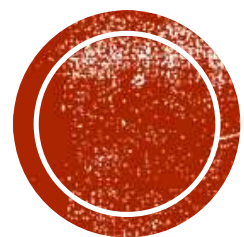
That's exactly
what you get



ANOTHER MAJOR BENEFIT: TYPES & SCHEMA

- GraphQL uses a strong type system to define capabilities of the API
 - Resource hierarchy as Type hierarchy
- The schema defines the contract between the client and the server
- Once there is an agreed upon schema
 - The front-end and back-end development can proceed independently





GRAPHQL CORE CONCEPTS



THE SCHEMA DEFINITION LANGUAGE (SDL)

- SDL for simple types
- Bang indicates 'required'

```
type Person {  
  name: String!  
  age: Int!  
}
```

```
type Post {  
  title: String!  
  author: Person!  
}
```



THE SCHEMA DEFINITION LANGUAGE (SDL)

- We can easily add relationships between types (similar to OOP)
- Lets see how we can add one-to-many between person and posts
- [] indicates collection

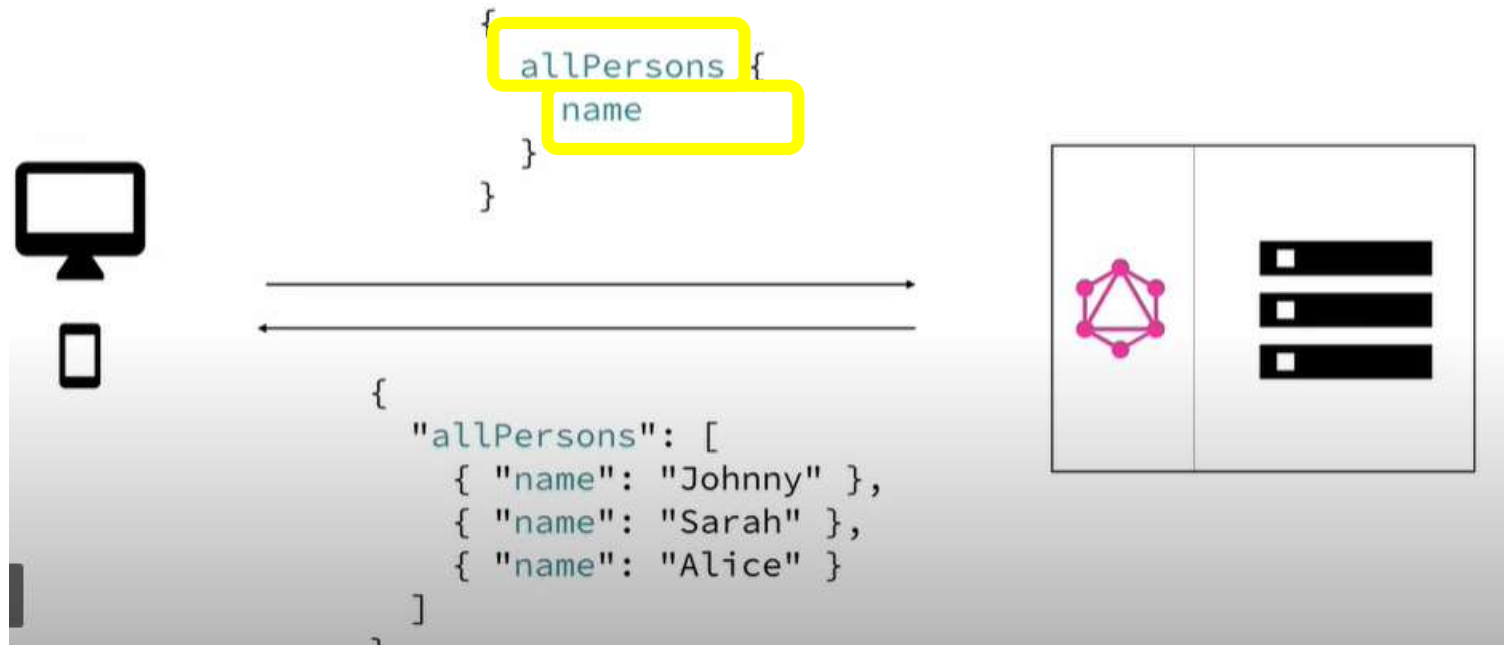
```
type Post {  
  title: String!  
  author: Person!  
}
```

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}
```



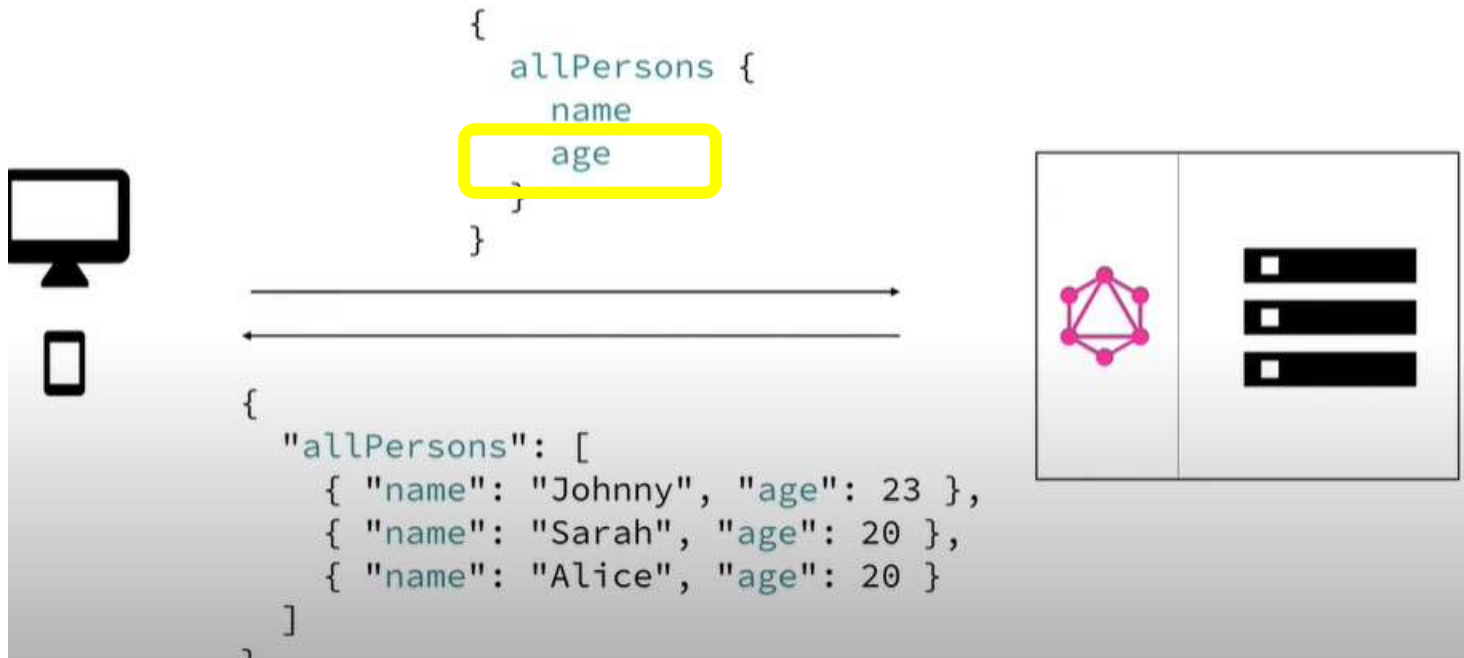
BASIC GRAPHQL QUERIES

- Root-field
- Payload



BASIC GRAPHQL QUERIES

- **Root field**
- **Payload** (modifying the payload slightly gives us more data)



BASIC GRAPHQL QUERIES

- Queries can accept **parameters**
- We can design parameters as we wish and support it in the backend



BASIC GRAPHQL QUERIES

- The beauty of GraphQL is the ability to support **nested Queries**
- Remember our person+posts schema

```
{  
  allPersons {  
    name  
    age  
    posts {  
      title  
    }  
  }  
}
```

```
{  
  "allPersons": [  
    {  
      "name": "Johnny",  
      "posts": [  
        { "title": "GraphQL is awesome"},  
        { "title": "Relay is a powerful GraphQL Client"}  
      ]  
    },  
    {  
      "name": "Sarah",  
      "posts": [  
        { "title": "How to get started with React & GraphQL" }  
      ]  
    },  
    {  
      "name": "Alice",  
      "posts": []  
    }  
  ]  
}
```



CHANGING DATA (MUTATIONS)

1. Creation of new data
2. Updating existing data (both full and partial updates)
3. Deletion of data



MUTATIONS

- Same syntactic structure as queries, but always starts with the mutation keyword
- Example of a **createperson** mutation

```
mutation {  
  createPerson(name: "Bob", age: 36) {  
    name  
    age  
  }  
}
```



MUTATIONS

- Same syntactic structure as queries, but always starts with the mutation keyword
- Example of a **createperson** mutation

```
mutation {  
  createPerson(name: "Bob", age: 36) {  
    name  
    age  
  }  
}
```

```
"createPerson": {  
  "name": "Bob",  
  "age": 36,  
}
```



MUTATIONS

- One common pattern is to use the ID GraphQL type for uniqueIDs

```
type Person {  
  id: ID!  
  name: String!  
  age: Int!  
}
```

```
mutation {  
  createPerson(name: "Alice", age: 36) {  
    id  
  }  
}
```



MUTATIONS

- Update Mutations does not require anything special
- You just pass in ID as one of the params, along with the params that needs to be updated



SUBSCRIPTIONS

- Another notable advantage of GraphQL is support to streaming Data
- Subscriptions represent a stream of data sent over to the client
- You subscribe to events and when that event happens the data you asked for is sent over

```
subscription {  
  newPerson {  
    name  
    age  
  }  
}
```

```
{  
  "newPerson": {  
    "name": "Jane",  
    "age": 23  
  }  
}
```

Note: Subscription is out of scope for us. We don't expect you to stream data. You are free to explore more on your own.



LET'S REVISIT SCHEMA

- Schema is simply a collection of GraphQL Types
- However (in convention) we prefer some typical root types, especially for APIs
- These are the entry points (Query, Mutation & Subscription).
- When you don't specify anything it defaults to Query

```
type Query { ... }  
type Mutation { ... }  
type Subscription { ... }
```



LET'S REVISIT SCHEMA

- Example of Query with just allpersons
- Example of Query with improved allpersons, where you can specify limit

```
type Query {  
  allPersons: [Person!]!  
}
```

```
type Query {  
  allPersons(last: Int): [Person!]!  
}
```



LET'S REVISIT SCHEMA

- Example of createperson mutations

```
type Mutation {  
  createPerson(name: String!, age: Int!): Person!  
}
```

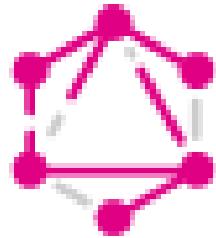


Final Full Schema

```
type Query {  
  allPersons(last: Int!): [Person!]!  
  allPosts(last: Int!): [Post!]!  
}  
  
type Mutation {  
  createPerson(name: String!, age: Int!): Person!  
  updatePerson(id: ID!, name: String!, age: String!): Person!  
  deletePerson(id: ID!): Person!  
}  
  
type Subscription {  
  newPerson: Person!  
}  
  
type Person {  
  id: ID!  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}  
  
type Post {  
  title: String!  
  author: Person!  
}
```



REFERENCE



HOW TO GRAPHQL

<https://www.howtographql.com/>

